Title Automated Test of Evolving Software

Name Hazel Anne Shaw

AUTOMATED TESTING OF EVOLVING SOFTWARE

Hazel Anne Shaw

A thesis submitted to the University of Luton, in partial fulfilment of the requirements for the degree of Doctor of Philosophy

October 2005

ABSTRACT: AUTOMATED TESTING OF EVOLVING SOFTWARE

Hazel Anne Shaw

Computers and the software they run are pervasive, yet released software is often unreliable, which has many consequences. Loss of time and earnings can be caused by application software (such as word processors) behaving incorrectly or crashing. Serious disruption can occur as in the 14th August 2003 blackouts in North East USA and Canada[1], or serious injury or death can be caused as in the Therac-25 overdose incidents[2].

One way to improve the quality of software is to test it thoroughly. However, software testing is time consuming, the resources, capabilities and skills needed to carry it out are often not available and the time required is often curtailed because of pressures to meet delivery deadlines[3]. Automation should allow more thorough testing in the time available and improve the quality of delivered software, but there are some problems with automation that this research addresses.

Firstly, it is difficult to determine if the system under test (SUT) has passed or failed a test. This is known as the oracle problem[4] and is often ignored in software testing research. Secondly, many software development organisations use an iterative and incremental process, known as evolutionary development, to write software. Following release, software continues evolving as customers demand new features and improvements to existing ones[5]. This evolution means that automated test suites must be maintained throughout the life of the software.

A contribution of this research is a methodology that addresses automatic generation of the test cases, execution of the test cases and evaluation of the outcomes from running each test.

"Predecessor" software is used to solve the oracle problem. This is software that already exists, such as a previous version of evolving software, or software from a different vendor that solves the same, or similar, problems. However, the resulting oracle is assumed not be perfect, so rules are defined in an interface, which are used by the evaluator in the test evaluation stage to handle the expected differences.

The interface also specifies functional inputs and outputs to the SUT. An algorithm has been developed that creates a Markov Chain Transition Matrix (MCTM) model of the SUT from the interface. Tests are then generated automatically by making a random walk of the MCTM. This means that instead of maintaining a large suite of tests, or a large model of the SUT, only the interface needs to be maintained.

1) NERC Steering Group (2004). Technical Analysis of the August 14, 2003, Blackout: What Happened, Why, and What Did We Learn? July 13th 2004. Available from: ftp://www.nerc.com/pub/sys/all_updl/docs/blackout/NERC_Final_Blackout_Report_07_13_04.pdf
2) Leveson N. G., Turner C. S. (1993) An investigation of the Therac-25 accidents. IEEE Computer, Vol 26, No 7, Pages 18-41.
3) LogicaCMG (2005) Testing Times for Board Rooms. Available from http://www.logicacmg.com/pdf/tracked/testingTimesBoardRooms.pdf
4) Bertolino, A. (2003) Software Testing Research and Practice, ASM 2003, Lecture Notes in Computer Science, Vol 2589, Pages 1-21.
5) Sommerville, I. (2004) Software Engineering, 7th Edition. Addison Wesley. ISBN 0-321-21026-3.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

This research was started during a Teaching Company Scheme between Plasmon and the University of Luton. I was employed on this scheme to investigate methods to improve software testing at Plasmon. During this time the research project developed into my PhD.

The research has concentrated on commercially viable techniques, dealing with problems that are often overlooked by other research within software testing.

I would like to thank a number of people and institutions for their help both formally and informally during this research project. Firstly I would like to thank my supervisors Diana Burkhardt and Alfred Vella for their continued and long-suffering advice and support; Angus Duncan for his help in providing a reliable contact within Luton University; David Golds and Mark Broadbent (both formerly of Plasmon) for providing the opportunity to start this research project; Plasmon for providing the software and equipment necessary to evaluate the feasibility of the research; The Institute for Manufacturing, Cambridge University for use of various facilities while writing up the thesis; and my husband, Andy Shaw for continued moral support and belief in my abilities. I would also like to thank the people with whom I have had informal chats over coffee or a drink who have provided encouragement to continue and complete this work.

During the research five publications have been made. These are in my maiden name of Curtis:

- "Development and Implementation of a System to Automatically Test Evolving Software". H. Curtis, D. Burkhardt, A. Vella. Keynote paper, OR43 conference, Bath, Sept 2001. ISBN 0 903440 261, pages 115-127

- "Automated Testing of Evolving Software Using an Oracle". H. Curtis, A. Vella, D. Burkhardt. OR42 conference, Swansea, Sept 2000.

- "Reverse Engineering Software Testing"; H. Curtis, A. Vella, D. Burkhardt. OR40 conference, Lancaster, Sept 1998.

- "Automated Test Suites from Reverse Engineering and Planguage"; H. Curtis, A. Vella, D. Burkhardt, M. Broadbent. Software Quality Week, San Francisco, May 1998.

- "Software Testing as an Aid to Quality Software Production", A. Vella, H. Curtis, D. Burkhardt, M. Broadbent. Total Quality Management, Yugoslav Association for Standards and Quality, No. 2, Volume 26, 1998, pages 480-485.

Finally, I would like to make clear that the research presented in this thesis is all my own work. Research and work that is not my own is clearly attributed (by use of references).

Two pieces of software are mentioned in this thesis: Alltest and Manager. Alltest is prototype software that I developed during the research project to demonstrate the feasibility of the proposed methodology to automate software testing. Manager is software developed by Plasmon and used as a case study for this research project.

AUTHOR'S DECLARATION

I declare that this thesis is my own unaided work. It is being submitted for the degree of Doctor of Philosophy at the University of Luton. It has not been submitted before for any degree or examination in any other University.

Signature: H. Shaw

(Hazel Anne Shaw)

Date: 30th March 2006

LIST OF ABBREVIATIONS

API	Application Program Interface
CVS	Concurrent Versions System
GUI	Graphical User Interface.
Lex	Lexical Analyser
MCTM	Markov Chain Transition Matrix
SUT	System Under Test
TAC	Test and Compare
TCD	Test Command Description.
Tcl	Tool Command Language
UOF	Unit of Functionality
VT	Variable Table
Yacc	Yet Another Compiler-Compiler

For definitions of these (and other) terms please see the Glossary on page 210.

CHAPTER 1 INTRODUCTION

The quality of software produced is a major concern to both software practitioners and users of software. One way to improve the quality of delivered software is to improve the effectiveness and efficiency of the testing process. However, though software testing is a key part of good software development practice, it is often not carried out well. In a survey of companies in the UK, Holland and Sweden 32% of respondents blamed poor testing on the drive to meet release deadlines with 72% saying that testing is often compromised in favour of development time. 85% of respondents said that there was poor availability of testing resources and capability while 78% said that skills are not available to test new developments (LogicaCMG 2005).

Despite the importance of software testing, much of the research in this area is concentrated on a very small subset of the problem, primarily test case selection (Bertolino 2003), and avoids looking at the bigger picture. The research presented in this thesis has taken an holistic approach to developing a software testing methodology.

This chapter starts by looking at commercial software testing, and the consequences of software failures. Section 1.2 gives an overview of this research. Section 1.3 examines the current knowledge that is relevant to this research. Section 1.4 gives the conclusions drawn from the literature review. Section 1.5 outlines the research aims. Section 1.6 explains the novel features of this research and finally, section 1.7 outlines the structure of the rest of the thesis.

1.1 COMMERCIAL SOFTWARE TESTING AND QUALITY

Testing of software is carried out for two reasons. The first is to find bugs in the software, or to confirm that a known bug has been fixed. The second reason is to establish the risk associated with the software. Software testing reduces the uncertainty about the perceived quality of a product (Marick 2005b). This enables a

commercial decision to be made about releasing the software, given the risk of software failure following release.

These two purposes are in conflict. On the one hand bugs need to be found. This is achieved by applying tests that are most likely to make the software fail. On the other hand a measure is needed of the software's reliability. This is achieved by applying tests that are a representative subset of the uses of the software. Two different sets of tests need to be applied, yet, in practice there is rarely the time or resources to apply two different testing strategies.

Mature knowledge is applied in engineering disciplines to obtain products with predictable quality. One difficulty with software development in practice is the maturity level of the knowledge of software testing techniques. Juristo *et al* (2004) have reviewed a series of empirical testing technique experiments. This review established that the maturity level of software testing knowledge is low. Which means that software engineers have to select testing techniques based upon intuition, current trends or the sales patter of testing tool vendors.

This section starts by looking at the consequences of bugs in software. Section 1.1.2 examines how software testing is incorporated into the software development life cycle in commercial organisations and section 1.1.3 discusses the people available to do testing. Section 1.1.4 examines test completeness. Finally, section 1.1.5 explores the project to implement CAPSA, accounting software bought by Cambridge University.

1.1.1 Software Failures

Computers and the software they run are pervasive. They affect every aspect of our lives. Yet released software is often unreliable and software that is failing can have further consequences. Almost everyone will have been affected by problems occurring in software, whether directly or indirectly. For example, many people who have used Microsoft's Windows operating systems and Office tools have faced

bugs in the software. The bugs can be minor (for example, problems with styles being applied in a word processed document) to major (for example, a crash that loses the last half hour's work). Such failures are a cost to business and take time and effort to resolve. The consequential lost earnings resulting from many small failures probably runs into millions or even billions of pounds on a national scale. However, some software failures have a much greater individual cost, resulting in damage to property, serious injuries or loss of life and major disruption. The rest of this section examines three such software failures

The most recent of these failures occurred on 14th August 2003 and resulted in an estimated 50 million people having no electricity in eight states in America and in two provinces in Canada (NERC 2004). This caused gridlock in cities as traffic lights failed. Some people lost water supplies because there was no electricity to pump the water. (Fox News 2003). A software failure was a significant factor in this blackout. Had a software system not failed it is likely that the blackout would not have occurred, or been much more localised in effect.

FirstEnergy uses software, an Energy Management System (EMS) to monitor its distribution of electricity. Part of the EMS is an alarm and logging system. This monitors the network, and raises the alarm if switch positions have changed or values that are monitored have exceeded or dropped below set limits. At 2:14pm this alarm system "stalled". This meant that problems on the electricity grid were not reported, making it appear that the electricity grid was operating normally (U.S.-Canada Power System Outage Task Force, 2004).

August the 14th was a warm day in North East USA. There were moderately heavy loads on the network as a result of air-conditioners running. Further loads were put on the network by a reactive generator, Eastlake 5, failing and electricity being imported into the Ohio area. This made voltage management in Ohio "more challenging" (U.S.-Canada Power System Outage Task Force 2004, page 27). This load and the warmth of the day caused the high voltage power lines to sag. Some of the lines touched overgrown trees causing short circuits and tripping the lines out.

This caused any electricity the lines were to carry to be diverted onto other lines in the grid. The increased load on these lines caused them to sag further, increasing the risk of more short circuits caused by tall trees.

Table 1 gives a timeline of some key early events. By 16:08 Northern Ohio had lost power. By 16:13 eight states in the USA and two provinces in Canada had no power.

Time	Event
13:31	Eastlake 5 trips out
14:02	A line trips out after making contact with an overgrown tree
14:14	FirstEnergy's EMS alarm and logging system stalls
14:27	Another line trips out after making contact with an overgrown tree
14:41	The server running the EMS alarm and logging system fails. The backup server takes over, but the alarm system remains in a hung state on the backup server.
14:54	The backup server fails
15:05	Another line trips out after making contact with a tree

Table 1 Timeline of early events on the Ohio electricity grid

GE Energy is the producer of the XA/21 EMS used by FirstEnergy in Ohio. Approximately 1 million lines of C/C++ code make up the alarm and event processing system. After eight weeks of investigation, GE Energy identified the bug as a race condition caused by two processes being in contention for the same data structure. The processes managed to write to the data location at the same time, causing corruption and allowing the alarm event application to get into an infinite loop. (Poulsen 2004).

Whether more testing or better testing would have found the error prior to the August 14th blackouts is unclear. However, straightforward techniques exist to avoid such problems with shared data in software. Silberschatz and Galvin (1999) devote an entire chapter of their book to process synchronization. It is also possible to increase the likelihood of detecting such an error by using multiprocessor machines when testing the software. Quad-processor systems are more than twice as likely to find such errors than dual-processor systems (Viscarola and Mason

1999). None of the published discussions on this software failure make it clear that such techniques were used.

Another software failure is the explosion of the Ariane 5 launcher on 4th June 1996. This was caused by an unexpectedly large value in a variable in the inertial reference system. On failing, the inertial reference system transmitted diagnostic information to the launcher's main computer. The main computer interpreted this as flight data and used it for flight control calculations. This resulted in a large correction being made for an attitude deviation that had not occurred. The result was the destruction of the Ariane 5 launcher about 40 seconds after takeoff. It is likely that the error would have been detected if thorough system testing had been completed using the inertial reference system (Lions 1996). The rocket and its cargo were uninsured. Estimates for the value of the cargo were around £500 million, although no official figures appear to be available.

The last software failure discussed here involves a computerised radiation therapy machine, the Therac-25. Between June 1985 and January 1987 in the USA and Canada there were six separate incidents where massive overdoses were given to six patients. These resulted in debilitating injury and three deaths (Leveson and Turner 1993). The accidents were caused by the software allowing the equipment to administer enormous doses. The Therac-25 suffered from many problems, which made identifying that a problem had occurred exceptionally difficult. For example, the user interface relied upon error message codes that did not indicate what problem the machine was experiencing. The machine frequently exhibited problems that the operators had become accustomed to. When a serious problem occurred and the machine behaved as "normal" (displaying an error code and pausing treatment) the operator did not see any difference from usual behaviour and continued treatment (resulting in multiple overdoses for some patients). There were many causes for the failure of the software. These include: poor documentation, poor design (for example, allowing non-synchronised access to shared memory) and inadequate software testing. There were at least two different errors that resulted in overdoses being given. Good software development processes, including extensive

module and system testing, should have found the errors or prevented them being made, prior to serious accidents occurring.

1.1.2 Software Development Practice

There are many models for the software development project life cycle, see Figure 1.

The traditional waterfall methodology allows no feedback into the development process. Incremental development has feedback from testing back into the code. However, there is no feedback back into the requirements and design stages. Evolutionary delivery (Gilb 1997a, 1997b) works on the principle that you cannot know everything, therefore you cannot possibly know all the requirements, design, costs, timing and risks. Delivery of a total project is broken up into 2% increments. The basic requirements are mapped out and then the product is designed. The project is planned with releases of the most important features first. Results from each incremental release can then be fed back into the requirements and design, allowing these to be refined as the project progresses.

The traditional waterfall lifecycle's strengths are the well defined and detailed specifications and designs that are produced early in the project. These allow developers to write code quickly and with few revisions. As a result defects are not introduced into the code because of many changes.

Conversely, techniques such as evolutionary delivery that emphasise lots of milestones, early beta testing and feedback from the customer, are very flexible. They allow a business to respond quickly to changes in customer requirements and market demands. Microsoft uses an approach that Cusumano and Selby (1996) call "synch-and-stabilise". Features are developed before they are fully understood. Changes to software are synchronised using a daily build process, then stabilised using testing. Customers are able to give feedback on features during the development process. This allows Microsoft to deliver what the customer wants.

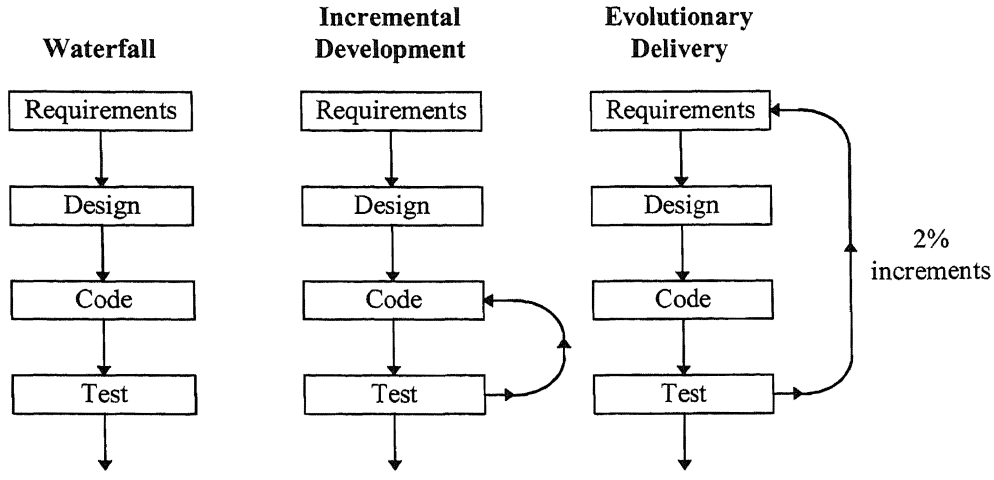


Figure 1 Examples of software development life cycles

Extreme programming (Jeffries *et al* 2001) takes the ideas of flexible, iterative development to the extreme. It is characterised by the customer being available to the development team at any time (preferably the customer should be with the development team on-site and full-time). The customer works with the developers to define the features. The customer selects the features they want to see in the project first (or next), having discussed with the development team how difficult each feature is to implement. Releases are made up of a number of small increments. Each increment is only two or three weeks long. In this way small releases are planned, one increment at a time. Another feature of Extreme programming is the use of Pair programming. This is reported to have many benefits. For example two programmers working together will produce more, and better, code than the same two programmers working separately. Finally, testing is emphasised in extreme programming. The customer must produce acceptance tests that are automated and the programmers must carry out unit testing (Jeffries *et al* 2001).

	India	Japan	US	Europe and Other	Total
Number of projects	24	27	31	22	104
Architectural specifications (%)	83.3	70.4	54.8	72.7	69.2
Functional specifications (%)	95.8	92.6	74.2	81.8	85.6
Detailed designs (%)	100	85.2	32.3	68.2	69.2
Code generation (%)	62.5	40.7	51.6	54.5	51.9
Design reviews (%)	100	100	77.4	77.3	88.5
Code reviews (%)	95.8	74.1	71	81.8	79.8
Subcycles (%)	79.2	44.4	54.8	86.4	64.4
Beta testing (%)	66.7	66.7	77.4	81.8	73.1
Pair testing (%)	54.2	44.4	35.5	31.8	41.3
Pair programming (%)	58.3	22.2	35.5	27.2	35.3
Daily builds (%)					
At the start	16.7	22.2	35.5	9.1	22.1
In the middle	12.5	25.9	29	27.3	24
At the end	29.2	37	35.5	40.9	35.6
Regression testing on each build (%)	91.7	96.3	71	77.3	83.7
Median output[1]	209	469	270	436	374
Median defect rate[2]	0.263	0.02	0.4	0.225	0.15

1. No. of new lines of code / (avg. no. of staff × no. of programmer-months).

2. No. of defects reported by customers in 12 months after implementation / total source KLOC. We adjusted this ratio for projects with less than 12 months of data.

Table 2 Results from survey of 104 projects (Cusumano *et al* 2003)

Cusumano *et al* (2003) have carried out a survey of 104 projects over companies in the US, Europe, Japan and India. This survey compared the different development practices prevalent in these areas and the quality of the software produced (quality was measured as number of defects found in a 12 month period following release per 1000 lines of code). In their paper, Cusumano *et al* note that the projects sampled are likely to represent better than average projects because good project management would have been necessary to allow respondents to complete their questionnaire. Table 2 summaries these findings.

As companies move towards using evolutionary development methods such as those advocated in extreme programming and evolutionary delivery, the requirement is for flexible methods to automate testing. Test automation should not rely on a specification that is set in stone. It should be possible to quickly and

efficiently develop new tests and alter existing tests. For automation to be useful and successful it needs to be flexible. It needs to cope with changing software, requirements, specifications and project deadlines.

1.1.3 Testing Resources

This section discusses the human resources (the people) available to carry out testing, their abilities and the time available to do the testing.

Testing is fragmented into different stages, for example: unit, module, integration and system testing. Usually unit testing is carried out by the developer. This is often informal, though it can be formalised by the building of small test harnesses to aid the testing of single functions. Scripts of tests can then be written and these are run to ensure that the functions work satisfactorily. Coverage tools can be used at this stage to show how thorough the testing has been. Component testing is also the responsibility of the developer. Integration testing is placed somewhere between the developer(s) and an independent test team.

System testing is invariably carried out by an independent test team. By the time system testing is underway a great deal of pressure is on the testing team to evaluate the product as quickly as possible. The number of dedicated testers can vary. For major software companies the ratio can be one tester to two developers (Kit 1995). However, most companies operate with poorer ratios of 1:3, 1:4 or above, and Kit observes that testers working in companies operating at 1:5 or above felt that testing was under resourced. Dustin *et al* (1999) suggest ratios of testers to developers of up to one tester to six developers (1:6), depending upon the development type.

Often this testing phase gets squeezed so that only a minimal amount of testing can be performed before the product is released. Only the discovery of a really serious problem or a large number of small problems is likely to force a halt on delivery of the software. Decisions are often made at this stage to ship with known bugs.

These bugs will usually be documented in a "release note" that accompanies the released software.

Generally the testers are skilled in finding problems in the software. However, with conflicting aims it is easy for testers to concentrate on showing the software works, rather than finding where it does not. This is an unconscious response. Often a new tester working on a product for the first time will find many more bugs than established testers and these bugs may be quite serious.

Testers are not software developers. This does not mean that testers cannot write any code, but their code development skills are usually much less well developed than those of a programmer. As such, tools that are designed for testing must take into consideration that any programming required should be as simple as possible. In such an environment tools to automate the system testing in an acceptable way are desperately needed.

1.1.4 Completeness of Testing

One problem when deciding the timing of a software release, is establishing when the software testing is "complete". Criteria to indicate when testing is complete may include: measured coverage meeting a specified target (e.g. 80% branch coverage); analysing the trend of bugs being found over effort in testing to reveal a decrease in the rate of bugs being found (implying less bugs to be found); or a statistical analysis of failures to give a measure of software reliability that must be met or exceeded. What all the criteria imply is that once testing is finished, the software is certain to still have bugs and will still periodically fail.

Weyuker (2004) says that measures such as code coverage are necessary, but not sufficient, for determining the completeness of the testing process. This is because important operations that the user will carry out regularly have not necessarily been selected as test cases. Instead, Weyuker proposes using the operational distribution

of the software as the basis for determining the testing progress and thus when testing is "complete".

1.1.5 Commercial Software Development Case Study

CAPSA is accounting software implemented for Cambridge University. The project to implement this software suffered from many problems that illustrate the effect of poor development processes on the quality of delivered software.

CAPSA had an original budget of £4.3 million. By July 2001 the project had cost over £9.1 million (Finkelstein 2001). Brealey (2000) described CAPSA as Cambridge University's "Millennium Dome". For six weeks following CAPSA's launch, the system did not work at all. After six weeks the system was working but "ten times slower than usual" and without all the required features (Brealey 2000). The following is a summary of the report on CAPSA's implementation (Finkelstein 2001).

The CAPSA project started life as a lightweight solution building upon Cambridge University's existing in-house accounting software. However, some way into this project a review was carried out and a decision made to buy an "off the shelf" accounting package. Following a requirements review, SAP and Oracle demonstrated their systems. Oracle was chosen as the preferred supplier, and the project began in earnest. Oracle was chosen despite a visit to the California Institute of Technology, where the demonstration system was not fully functional and warnings were given about the research grant module.

The CAPSA project suffered from many project management problems. For example, the University did not manage the project closely, with project management being provided by a contractor. The staff working on the project came from Oracle, various consultancies and seconded university staff. Morale was very low and personal relationships between the groups were exceptionally poor. There were difficulties with who was responsible for the management of the project.

Communication between the various parties was also exceptionally poor. During the project some key University staff took early retirement. Many of these staff had expert knowledge of the university systems and accounting practices.

A major change needed to be made to the Chart of Accounts, a key component of any accounting package. This requirement only became apparent some way into the project. Part of the implementation of CAPSA was to configure Oracle's system to use in the University. During this configuration, a great number of bugs were found, both minor and major. The time taken to deal with these bugs resulted in the project schedule slipping. As a result the testing phase became very compressed, with little effective integration testing, no regression testing and no "volume" testing (that is testing the system with large quantities of data and users). User acceptance testing was started very late in the project, and also suffered problems. As a result the acceptance testing was not completed, and the users did not sign off on the testing.

Training for users of the system was inappropriate. Computer literate staff were mixed with staff who had little computer experience. The system used for the training was unstable, with lockups and crashes. As a result the training did not meet anyone's needs.

A go-live date of August 2000 was proposed early in the project's life. This date was never changed. Due to schedule pressures, the go-live was not a gradual rollout, but a big-bang, with the whole University moving over to the system at that date. In addition, the existing accounting system was not used in parallel with CAPSA. CAPSA was unusable for the first six weeks after go-live. A review of the state of play at this point concluded that it would take eleven people six months to deal with the critical problems and stabilise CAPSA.

One of Finkelstein's conclusions is that Oracle supplied a poor product where parts were only marginally fit for purpose, and that such poor quality is a feature of the software industry as a whole.

1.2 Overview of Research

There are three distinct phases that are carried out when testing a software product. These are generation of tests, execution of tests and evaluation of results. This research has developed a methodology that addresses all three phases and is applicable in commercial software development environments. To do this a number of problems needed to be solved.

Firstly, there appears to be a problem where methods to automate component/module and system testing on large systems are not available.

Secondly, many software development organisations use an iterative and incremental process, known as evolutionary development, to write software. Following release, software continues evolving as customers demand new features and improvements to existing ones (Sommerville 2004). This software evolution means that automated test suites must cope with continuous changes to the specification and requirements of the system under test (SUT). The test suite developed must also remain usable throughout the life of the SUT. However, maintenance of a test suite is very difficult, because even little changes in the SUT may result in big changes being made to the test suite.

Finally, it is necessary to establish when a test has been passed or failed by the SUT. This is known as the oracle problem (Bertolino 2003).

The methodology developed uses predecessor software as an oracle. This is software that already exists. It may be a direct ancestor, such as a previous version of evolving software. Alternatively, it may be an indirect ancestor, which is software written independently from the SUT, for example from a different vendor, that performs the same, or similar, function. An interface is written that defines "units of functionality" (UOFs) for the SUT. The UOFs correspond to the behaviour of the SUT broken up into small functional units. A Markov Chain Transition Matrix, which models the SUT, is automatically built from the UOFs in the interface. Tests, which are sequences of calls to the UOFs, are generated by

randomly walking this matrix. The comparison phase uses rules that allow the comparator to deal with the differences between the outputs from the oracle and the SUT.

A test system (called "Alltest") has been designed and implemented by the author to demonstrate the methodology developed during this research. It generates tests, runs the tests and evaluates the result of running the tests. Alltest and the methodology it implements are described in Chapter 2. A second system is also needed, this being the System Under Test (SUT). The SUT chosen is commercial software called Manager, and is described in Chapter 3.

This research shows that the use of predecessor software as an oracle is a viable solution to the oracle problem.

However, the oracle may not be perfect, so this research demonstrates that rules can be used to handle the differences between the SUT and oracle system.

The research also demonstrates how a Markov Chain Transition Matrix that models the SUT can be automatically built from an interface that defines functional inputs and outputs to the SUT. The tests are then created by taking random walks through this matrix.

1.3 CURRENT KNOWLEDGE

Software testing, like any scientific or engineering discipline, has its own set of specialist terminology. Colloquially "problems", "faults", "errors" and "failures" in software are often known as "bugs". To be absolutely correct there is a distinction between the terms "fault", "error" and "failure" (Bertolino 2003). If, when testing a program, it produces an unexpected output, this may be a failure. If it is a failure then the originating cause of this failure is a fault (for example in the program code). A fault can remain undetected in the software until some condition or input causes the fault to be activated. When the fault becomes activated, it will produce

an error. If this error is propagated through the software it can become an observable failure.

The literature is inconsistent about the definition of "fault", "error" and "failure". For example, Marick (1995) defines an "error" as a mistake made by a developer (for example, a misunderstanding of a specification). An "error" leads to a "fault" written into the program code. A "failure" may occur when faulty code is executed. For the purposes of this thesis the terms "error" and "bug" shall be used to broadly encompass the concept of a mistake made in the software, and the resulting failure of the software.

Various testing techniques are used to reduce the effort needed to test software while increasing the number of errors that are found in the software. The techniques that are applied to testing software can be either totally manual or aided to various degrees by computers. These techniques can be split into two types, black-box and white-box. Black-box techniques treat the software (or component) as a black-box with inputs and outputs, but no understanding of what is happening within the black-box. White-box techniques analyse the structure and logic of the software. This is used to monitor the thoroughness of testing. It is also used to guide the selection of appropriate test data to cause previously unexecuted areas of code to be run (Myres 1979; BCS 2002).

Bertolino (2003, page 2) defines software testing as:

> "... the **dynamic** verification of the behaviour of a program on a **finite** set of test cases, suitably **selected** from the usually infinite executions domain, against the specified **expected** behaviour."

The bold terms can be explained further (Bertolino 2003).

- **dynamic**: this makes it clear that testing requires the program to be run, as opposed to static checking of software as discussed in section 0.

- **finite**: exhaustive testing is not possible. The total number of possible inputs is practically infinite, and the time and resources needed to test the software would make it impossible to use all possible inputs. Bertolino says that exhaustive testing is not possible even for the smallest program.

- **selected**: test criteria should be used to select the test cases. A great deal of research has been carried out on different techniques to select test cases. Most of these look to limit the number of test cases while maximizing the likelihood of detecting an error, thus increasing the testing efficiency. Test selection techniques are discussed in section 1.3.2

- **expected**: a test case should include not only the input data, but also the expected outcome. The behaviour of the system under test can be checked against user expectations (validation) or against a specification (verification or conformance testing). Determining whether a test has passed or failed is known as the oracle problem. This is discussed in section 1.3.3.

The rest of this section reviews the current knowledge and examines the state of the art for testing software and producing quality software products. It starts by discussing non-testing methods for detecting errors in software. This section then examines: test case selection (including test case generation), how to evaluate the outcome of the tests and how to manage and execute the tests. This section finishes by examining software testing in commercial environments.

1.3.1 Alternative Quality Assurance Techniques

Dynamic testing of software is not the only way to prevent or detect errors in software. This section examines some other approaches that can be used.

So, Sha, Shimeall and Kwon (2002) examined six methods used to detect faults in software. The methods examined were voting (as in N-Version programming), testing (requirements and design based testing with the aim of 100% requirements

coverage), self checks, code reading, data-flow analysis and Fagan inspections. The methods were checked against five programs written by senior computer science undergraduates. The programs were battle simulations written in Pascal with between 1500 and 7500 lines of code (including comments). So *et al* (2002) found that of the six methods, three of them, Fagan inspections, voting and testing, discovered the majority (77%) of the total number of bugs found in the programs. Of these bugs 70.7% were found by only one of these methods, showing that these three methods are complementary when used together.

"Static" Techniques for Evaluating Software

Static techniques involve examining the source code for errors. Desk checking (examining ones own code) and peer reviews (examining a colleagues code) are informal methods. However, code inspections and walkthroughs are formal processes where a meeting is held and a group (usually consisting of three to five people, including the programmer) work through the code looking for errors. A checklist is often used to help with the detection of errors. Roles such as moderator and secretary are usually assigned to the participants (Myres 1979).

Static analysers (such as LINT for C) can be used to search through the code looking for common errors such as referencing uninitialised variables. These should detect potential errors that a compiler may miss. However, many modern compilers do a good job and produce warnings for code that may not function as the programmer intended (for example, in 'C' it is possible to use an assignment within a condition statement).

Inspections are not just limited to code; they can be carried out on all project documentation. If an error is detected early in the software development process (for example, within the specification), then correcting that error will cost significantly less than if it is found late in the process (Gilb and Graham 1993).

Formal Methods and Proofs

Formal methods and proofs rely on the application of mathematical techniques. Finney (1996) showed that even with training in discrete mathematics and the Z notation, many computer science students had difficulty in understanding even simple specifications. Though Finney says these results should be treated with caution, it does raise concern about the ability of software engineers to use formal methods when developing software. Therefore the emphasis of this research is on techniques that may be applied within most industrial software development environments, and avoids techniques using formal methods.

Code Metrics

Writing code with a certain style may reduce errors. Code metrics can be applied to code and used to guide the development of software. For example, a guideline often exists that a function should be printable on a single page of A4 paper. This means that in practice a function should not exceed about eighty lines of code (LOC). This is because it is believed that functions exceeding this size are more likely to have bugs and are difficult to maintain. However, measures such as LOC are very poor. They are affected by the style a programmer uses to write code and do not correlate across programming languages.

An alternative code metric is Halstead's length measurement (Halstead 1977), in which a count is made of the operators and operands in the program. This measure correlates between programs written in the same language, but does not correlate to different languages that have different sets of operators. A code measurement that correlates between different styles and different languages is the McCabe Cyclomatic Complexity Measure (CCM) (McCabe and Watson 1994). This is calculated by counting the decision constructs in a piece of code. It is usually calculated for each function. McCabe and Watson suggest that the CCM for a function should not be above ten because the number of errors in the function jumps

when the CCM is above this level. However, this restriction is disputed in practice (Vinter 1997) and should be considered a guideline and not a hard and fast rule.

1.3.2 Test Case Selection

Software testing is exploratory. When testing software we are trying to discover errors in the software that we do not know are there (Armour 2004). For this reason, techniques are required that allow testers to select test cases, where the test cases have a high probability of finding an error. This section covers test case selection methods, both manual and automated. It starts by looking at black-box and white-box techniques in general. It finishes by examining two black-box approaches that use models to generate the tests. These are using a Markov chain usage model (page 26) and using a UML model (page 29).

Black-box Techniques

Black box testing techniques do not use the source code of the software to guide testing. These techniques use the specified inputs and outputs of the software system, component or function being tested to guide the selection or creation of suitable test cases.

Error Guessing is the process of creating test cases in an ad-hoc way based upon instincts about where errors may occur (Myres 1979). It is a useful supplement to the other more structured testing methods. A good tester uses their experience and unconscious knowledge to select tests that are more likely to find errors (Armour 2004).

Equivalence Partitioning involves dividing the input space for the program into classes of data which according to specification are treated identically. The equivalence classes are identified by taking each input condition and partitioning it into two or more groups so that there are valid equivalence classes for valid inputs and invalid equivalence classes for all other possible (erroneous) inputs.

Equivalence classes must not overlap so any such cases must be reduced to separate and distinct classes (Myres 1979).

Once the equivalence classes have been identified they can be used to define a set of test cases as follows:

1. Assign a unique number to each equivalence class.

2. Write a new test case. Each test case should deal with as many valid equivalence classes as possible. Continue creating test cases until there are test cases that deal with all valid equivalence classes.

3. Write a new test case that deals with a single invalid equivalence class for each invalid equivalence class.

Partition Testing is a more general term describing such techniques. Essentially, the program's input domain is partitioned into subdomains. The test hypothesis is that for every point in the same subdomain, the program will either succeed or fail. Therefore, only one point in the subdomain needs to be checked (Bertolino 2003).

Boundary Value Analysis is carried out once the equivalence classes have been defined. The boundaries are located and test cases are specified directly on, above and beneath these boundaries. Boundary value analysis focuses on an area with a potential for a high yield of errors.

Cause-Effect Graphing is a technique to aid the selection of a set of high yield test cases. A cause-effect graph is a formal language. It is the same as a digital logic circuit, but uses a simpler notation than standard electronics notation. A natural language specification will be translated into a cause-effect graph (Myres 1979).

Statistical Testing starts with an analysis of the expected and known uses of the software. A model of the software is built. This model can then be used to select a random sample of test cases. This sampling approach allows statistical analysis of the software, enabling potentially useful predictions about the reliability of the

software to be made. Different types of model can be used both to guide the selection of a random sample of tests and for the subsequent analysis of the reliability of the software (Poore and Trammell 1998). An example is the Markov chain usage model (see page 26).

Another use of statistical models is to predict the fault density (bugs per thousand lines of code) and location of bugs in software. Ostrand, Weyuker and Bell (2005) have developed a negative binomial regression model that analyses the fault and change history of the software under test. This model can then predict the files that contain the largest number of faults in a release, and allow regression testing to be targeted at the areas which will have the highest payback. The authors are currently undertaking further work to analyse the effect of applying the model over a series of software releases and to automate the process so that a high level of statistical expertise is not required by software engineers to apply this technique.

Experiment design covers a number of different but related techniques. These techniques derive from engineering design of experiments where the effect of parameters in a system needs to be evaluated. The idea of experimental design is to minimise the number of tests, while producing a set of experiments that allows the effect of changing each parameter, and parameter interactions to be observed. One technique makes use of a mathematical construct called an Orthogonal Array. Taguchi developed a family of matrices (which are mathematically identical to orthogonal arrays) to be used in a variety of situations (Ross 1988). Using such a method, it is possible to construct an experiment analysing, for example, 13 variables each with 3 different settings (or levels) in only 27 trials. In contrast, an attempt to test all possible combinations, a full factorial design, would need approximately 1.6 million experiments (3^{13} = 1,594,323). The experiment results are analysed with a statistical technique called "Analysis of Variance" (ANOVA) (Ross 1988; Bicheno 1998).

Various researchers have applied such techniques to software testing. For example orthogonal arrays have been applied to the testing of an ADA compiler (Mandl

1985). Mandl concluded that a set of tests equivalent in size to a randomly selected set of tests could yield useful information equivalent to exhaustive testing (where exhaustive testing means taking each variable in turn and testing at all possible levels). Taguchi methods have also been applied to unit testing (He *et al* 1997). The Robust Testing Method (Phadke 1997) also uses Orthogonal Arrays. A case study is reported where the Robust Testing Method was applied at AT&T for system testing. In this case a drastic reduction in the number of tests required was seen, along with an increase in the number of bugs found. The study reported that following the correction of the reported bugs and the subsequent release of the software, no additional bugs were found in the area tested with this method after two years in the field.

A similar method is combinatorial design (Cohen *et al* 1996; Cohen *et al* 1997). This is applied to the automatic generation of test cases. This method relaxes the requirement for the testing to be balanced (or orthogonal). Instead it concentrates on producing a test plan that covers all pair-wise, triple or n-wise combinations of test parameters. However, in removing the requirement for orthogonal tests, the post-test analysis that is a valuable part of the Taguchi method cannot be applied. This means that the testing is primarily used for finding errors. Further experiments are needed once a bug is found to establish more information about which parameter(s) are interacting to cause the problem. The combinatorial method is claimed to be usable on systems with large numbers of parameters, as the number of tests produced is much fewer than with Taguchi methods. Cohen *et al* (1997) claim that a combinatorial test over 100 binary parameters will need just 10 tests, compared with 101 tests for Taguchi methods.

It is unclear how effective the combinatorial method is at finding bugs. For example, Cohen *et al* (1997) report an average of two code and four requirements bugs over nine modules of the system being tested. Each of these modules has from 1000 to 2000 lines of C code. If an average of 1500 lines of code is assumed, then this means that one error was detected for every 250 lines of code. Beizer (1990) suggests that a typical frequency would be one to three errors for every 100 lines of

code, dropping to one error in every 1000 lines of code following testing and debugging. Cohen *et al* (1997) say that the modules they tested using the combinatorial method had already been tested. However, it is not clear how thorough that previous testing had been and how many errors remained to be detected in the code. If the previous testing had been very cursory, then the number of defects found with the combinatorial testing method appears to be quite low.

White-box Techniques

White-box testing techniques use the structure of the software code to inform the selection of test cases and to enable measures to be made on the amount of testing completed. However, there is a fundamental problem with using a program's source code to guide the selection of test cases. The program's code is being used as the reference model, therefore it will be impossible to detect faults such as missing functionality, because the reference model and the program under test are the same (Bertolino 2003).

Path-Based Testing selects test cases to execute specific paths through the software code. A path is a sequence of instructions or statements executed through the program (Beizer 1990). Paths are selected through the program with the aim of meeting some criteria. Examples of criteria are "execute all statements at least once" and "execute all branches at least once". In practice it is very difficult to select inputs that will force the program to execute a specific path. Strategies to do this include symbolic execution and genetic algorithms.

One example that uses symbolic execution is CSET (C Symbolic Execution Tool) (Dillon and Meudec 2004). CSET uses symbolic execution, logic programming and constraint logic programming to generate test cases that fulfil coverage criteria. Dillon and Meudec have tested CSET on functions from industrial embedded code and obtained path coverage measures from 16% to 59%. They conclude that this is an improvement on previous systems, but more work is needed.

An example that uses a genetic algorithm is GADGET (the Genetic Algorithm Data GEneration Tool), which employs dynamic test generation (Michael *et al* 2001). The source code of the program is instrumented so that information can be collected as the program is run. This information is then analysed and the test inputs are adjusted so that the test execution is gradually moved towards satisfying a test adequacy criterion.

Coverage Analysis is an alternative to path-based testing. Instead of selecting tests that force a specific path to be executed, coverage analysis is used to monitor how well the testing is exercising the code. Examples of coverage metrics are: function, statement, branch, condition, condition/decision, multiple condition/decision and path.

The process for using coverage analysis is as follows: A coverage analysis tool parses the source code and outputs instrumented code. This is the original code with added code (called probes). These probes allow the coverage analysis tool to establish when a statement, line or branch has been executed. Next, the instrumented code needs to be compiled. Once the software is compiled the tests can be run against it. Data are gathered indicating which parts of the code have been executed. An analysis can then be run on the data gathered, turning the collated data into coverage information. A tester can use this information to identify areas of testing that are weak and add new test cases as appropriate. Examples of coverage tools are GCT (Marick 2005a), BullseyeCoverage (Bullseye 2005) and JaBUTi (Vincenzi *et al* 2005). JaBUTi is unusual in that it instruments the Java bytecode instead of the Java source code. This enables coverage analysis to be carried out on third-party components, and on the final build of the software; something that cannot be done with coverage tools that instrument the source code.

One problem with coverage analysis is that the instrumented code will run slower than the original source code. Some coverage tools allow coverage information to

be targeted on specified areas of the software. This alleviates the speed problem and allows analysis to be targeted on recently added or changed areas of software.

Many companies specify coverage criteria that should be met through testing (e.g. 85% branch coverage). However, a linear increase in code coverage will result in an exponential increase in errors found (Williams *et al* 2001). If the probability of the software under test containing no errors is 0.001, then at 85% code coverage the resulting code quality is 35% (implying only 35% of the bugs have been found), whereas at 95% the code quality is 71%.

Mutation testing is another testing strategy that is used to evaluate the effectiveness of a set of tests, and select further suitable test cases. Mutation testing requires small changes to be made to the program to create a set of "mutant" copies. The original program and a mutant are then executed with generated input data. When a difference is found between the mutant and the original program, the input data that detected that difference is recorded, and the mutant is deleted. The process then continues with the next mutant. If the tests cannot distinguish between the mutant and the original program, then new test cases need to be written. There are two types of mutation testing, weak and strong. For strong mutation testing the difference between the mutant and the original program is detected in the output of the programs, whereas for weak mutation testing the difference is detected in the internal states of the programs (Voas *et al* 1993).

Error seeding is a fault-based, statistical testing method. It involves the deliberate insertion of errors into a program. The program is then tested and the number of original and seeded errors recorded. From these numbers an estimate can be made of the total number of errors that are in the program (Myres 1979).

Extended Propagation Analysis is another fault-injection technique similar to error seeding. Artificial hardware and software faults are simulated to establish how the software will behave under anomalous conditions. The aim is to establish unacceptable outcomes and estimate how "failure-tolerant" the software will be in

real use (Voas *et al* 1997). One example of a tool that supports fault-injection is "Verifier". This is freely available to Windows device driver developers on Windows 2000 and XP (OSR 2000). Verifier conformance checks device drivers and can also simulate low resources. Low resource simulation means that system calls that result in resources or memory being allocated are made to fail randomly for the driver being tested. This can enable the device driver writer to establish how robust the driver will be under low resource situations. This is a non-intrusive technique for the injection of external faults that can be carried out on the final version of the driver.

Testing with a Markov Chain Usage Model

There is some research within statistical software testing that uses Markov Chains to model software and its expected use by customers. Markov chains were originally proposed as a suitable vehicle for modelling the usage of software and for use in software testing by Whittaker and Poore (1993) and Walton, Poore and Trammell (1995).

There are some useful measures that can be obtained by using the Markov chains. For example the number of states necessary until a specific state can be reached, or the expected number of states before the termination of the software. The statistical analysis of the testing results can be expanded by the use of two Markov chains (Whittaker and Thomason 1994). The first Markov Chain is the usage chain that encodes the model of the system under test and drives the selection of test cases. The second Markov Chain is the testing chain that is built while the software is run. Instead of probabilities (as in the usage chain) the testing chain records frequencies of the transitions as they occur. These frequencies are converted into probabilities whenever any computation needs to be carried out on the testing chain. If a failure occurs, then a new state is added to the testing chain and the transitional frequency is set to 1. Further transitions into this failure state result in the transition frequency being incremented. This technique has been further developed, by using a testing chain for every different build of the software that is tested. These cumulative

testing chains allow a predicted testing chain for the next build of software to be produced. This predicted testing chain can be used to forecast the reliability of the next build of software (Whittaker *et al* 2000).

A Markov chain usage model can also be used to estimate the probability of failure in a given number of steps from a known state or the overall probability of failure from any starting state. This is achieved using a Bayesian model applied to the individual arcs in the Markov chain (Prowell and Poore 2004).

Unfortunately creating the Markov chains requires a large amount of manual work, which makes the technique unrealistic in industrial software development environments. The following steps are carried out manually to create a valid usage model:

1. Define the structure of the model (that is the states and the arcs between the states, without any probabilities assigned to the arcs).

2. Establish the several hundred transition probabilities (or relative frequencies).

3. Compute usage statistics for the chain including the usage profile.

4. Review the analytical values relative to requirements, specifications and testing plans.

5. Revise transitional probabilities (or relative frequencies) of the model in order to improve the realism of the analytical values calculated from the model.

An improvement can be made by generating the transitional probabilities (step 2) from a set of constraints (Poore *et al* 2000; Walton and Poore 2000a). There are three activities that make up the approach:

1. Determine test constraints: these can be structural or usage. Structural are those that follow immediately from the directed graph form of the usage model. For example, some states will only have one transition out of them ($p_{i,j} = 1$) or will

have no transition ($p_{i,j} = 0$). Usage constraints define values for particular transitions or relationships between sets of transitions.

2. Define appropriate objective functions. These are the test objectives. They allow, for example, the test manager to specify that certain areas of the software may need more testing than other areas.

3. Generate transition probabilities. Use standard mathematical programming techniques to determine a solution for the system of equations defined in (1) and (2).

Over time these equations can be altered as more information is learnt about the usage profile of the system or the test objectives change. The transition probabilities can be regenerated as needed using appropriate automated tools.

According to Walton and Poore (2000a) this process fits within an incremental software development environment. The model can develop as the software is developed. The constraints and objectives can change as required by the current phase of the software development. However, there is still a lot of work required to build the structure of the usage model, which is also likely to change over time. In addition there is no discussion about what tools are used to support this process, nor any about how the test execution is to be evaluated. For this technique to work, maximum automation is required to allow a large enough number of test cases to be run. This is because the test generation is stochastic and Beizer (1990) shows that random testing requires a larger number of test cases to be executed to find a similar number of bugs in the system under test when compared to branch testing and all-uses testing.

Markov chains have been used in other ways in software development and testing. For example, they can be used to provide a measure of complexity of a software specification and to provide a measure of the coverage achieved by testing (Walton and Poore 2000b).

Markov chains have also been used to stress-test telecommunications systems (Avritzer and Weyuker 1995). In this case a Markov chain was used to generate a set of stress-tests. The Markov chain used was a "birth-death process" chain. This is a chain where each state has just two neighbouring states. At each state it is only possible to transition forwards into a new state, or backwards into a previous state. The Markov chain modelled the number of calls of particular types, each transition occurring when a call was received (transition forwards into a new state) or dropped (transition backwards to a previously visited state).

None of the above work has discussed how the Markov Chain Usage Model should be represented. The Model Language (TML) addresses this problem (Prowell 2000). TML is a language that has been developed to:

- be simple (so that software engineers can quickly start to use it).
- separate structure from usage statistics.
- be extendable.

In the above work the primary aim of using a Markov chain usage model is to enable statistical inferences to be drawn from the testing results, allowing an estimate to be made for the reliability of the software being tested. In contrast, a Markov chain has been used in the research presented in this thesis to enable control of the stochastic test generation phase.

Using UML to Generate Tests

The Unified Modeling Language (UML) is a specification language for object oriented development that is widely used in industry (UML 2005). It is supported by readily available software such as Rational Rose (IBM 2005a). This widespread use in industry makes it an ideal candidate to use when testing software. A subset of the UML has been used as the basis for automatic test generation (Cavarra *et al* 2004). The diagrams used are class, state and object diagrams.

These diagrams allow the SUT to be described at an appropriate level of abstraction.

Using UML is a viable approach in software testing. However, Cavarra *et al* (2004) indicate some problems with the process. UML models developed as part of the design process may be too detailed to produce a useful suite of tests. This is because the underlying state space would be too large. If code is presented without UML designs then reverse engineering the code to produce the UML models also produces models that are too detailed or too closely related to the final implementation.

1.3.3 Evaluating Test Results

A great deal of research has been carried out into how test cases should be selected or generated (see section 1.3.2). However, research into how the outcome of a test is to be evaluated is much less comprehensive. There are two main approaches being researched: formalism and embedded test code (Whittaker 2000). Formalism uses formal specifications to provide the expected results. However, industry generally does not use formal methods (Glass 2004). The second approach is to use embedded code. One way of doing this is to write multiple versions of key routines. Each implementation is executed and the results evaluated. If the results are the same then there is unlikely to be a bug.

This section looks at these methods, and some related techniques. It begins by defining the "oracle problem".

Oracles and Their Classification

Determining whether a test has passed or failed is known as the "oracle problem" (Bertolino 2003). An oracle is something that decides whether the program has behaved correctly (or otherwise) for a particular test. According to the dictionary (Collins 1998) an oracle is: "a prophecy often obscure or allegorical revealed through the medium of a priest or priestess at the shrine of a god." Another

definition from the same dictionary is: "a statement believed to be infallible and authoritative". The second definition starts to get to the source of the oracle problem. It is assumed that for every test run, the oracle can say authoritatively whether the test has succeeded or failed. In reality, the oracle that is available is not always able to do this. Oracle coverage (Bertolino and Strigini 1996) is defined as "the probability that it [the oracle] rejects a test (on an input chosen at random from a given probability distribution of inputs) given that it should reject it". A perfect oracle would have oracle coverage equal to one. However, most oracles are not perfect.

The form of the oracle is another aspect of the oracle problem. The most basic oracle is the human oracle. Having read the requirements and the specifications, the human oracle examines every test run and determines if the test output is a pass or fail. However, human oracles are fallible. It is an exceptionally dull job and mistakes are easy to make. It is also possible for bias to creep in and tests to be passed because the human oracle wants (subconsciously) the program being tested to pass. The human oracle is also a severe bottleneck. It is possible to manually check the outcome from a handful of tests; it is not possible when hundreds or even thousands of tests have been run as part of an automated test strategy.

Usually an automated oracle of some form is required. However, an oracle may not exist. For example, how can a program be tested that produces the n^{th} digit of π? One solution is for the tester to find a "pseudo-oracle" (Weyuker 1982). The program can be tested for low values of n, which are known and can be easily verified. If these are generated correctly, the tester then assumes that the higher values of n will also be generated correctly. An alternative is to use an approximation to establish how plausible the result is. If the program produces a result that is nowhere near the approximation, then the tester can assume that the result is wrong (or at least implausible) even though the correct solution is not known. A "pseudo oracle" is also known as a "heuristic oracle" (Hoffman 1999).

The following are different classifications of oracle types (Hoffman 1999):

- true oracle: independent generation of all expected results
- heuristic oracle: verifies some values as well as consistency of remaining values.
- sampling oracle: makes a selection of inputs and results to check. Values not in this sample are not checked.
- consistent oracle: verifies current run results with a previous run (regression testing)
- no oracle: results produced from testing are not checked.

If an oracle is not available, then the only testing possible is robustness testing. JCrasher (Csallner and Smaragdakis 2004) is an automated testing tool for Java code. JCrasher tests the public interfaces of Java objects, randomly combining the methods to create data and states that are type-correct, but may reveal bugs. JCrasher is explicitly looking to exercise the software until a Java exception is thrown. However, without an oracle it is not possible to find bugs where the program inputs do not lead to the correct outputs.

An alternative classification of oracle types is provided by Hoffman and Strooper (1991). They define five types of oracle:

- Type I: Manual examination of each output from each test run.
- Type II: Manual generation of each output once, to be compared automatically to the output from each test run.
- Type III: Programs which generate (x,y) pairs, where y is the correct output from input x.
- Type IV: Programs which generate the correct output y for any input x.

- Type V: Programs which determine the correctness of any input/output pair (x, y).

Hoffman's (1999) classification of oracles is different from Hoffman and Strooper's (1991) classification. However, Hoffman and Strooper's (1991) types can be used to classify Hoffman's (1999) oracles. That is: a "true" oracle is a type IV oracle; a "sampling" oracle is a type III oracle; a "heuristic" oracle is somewhere between a type III and type IV oracle, where the type IV part is only approximate and the "consistent" oracle is a type IV oracle.

The following discussion looks at different ways that researchers have tackled the oracle problem. The different oracles discussed can all be assigned a type from Hoffman and Strooper's (1991) classification.

N-version Testing

N-version programming is an approach that is usually applied to fault-tolerant software. Three or more versions of the software (or a critical module) are produced independently. "Independently" may mean that different software engineering practices, designs and programming languages are used for each software version to eliminate common errors. All versions are then run. If the output produced does not agree, an error is likely to have occurred in one (or more) of the versions. These independent programs may be run as a majority voting system, such that if a majority agree on the result, then that is the result given. Improvements can be made to the system in the field by logging disagreements and investigating and correcting later. Although common errors should have been eliminated, such errors can still occur, but theoretically with a much lower incidence than a single instance program. However, research suggests that programmers working independently are likely to make similar (though not identical) mistakes that can lead to failures in the same area of the program (Brilliant *et al* 1990). There are coincident errors that can still lead to a failure of the system. N-version programming where N=2 is called dual programming. In

this case it is impossible to have majority voting, because it is unclear which version may be correct when a disagreement occurs.

The approach of N-version (or dual) programming can be applied to the oracle problem for testing software. This approach is sometimes called "back-to-back" testing (Vouk 1990) or *M*-mp testing (Manolache and Kourie 2001). Using Hoffman and Strooper's (1991) classification, such testing is using a type IV oracle.

A dual programming approach to software testing that aims to automate the analysis phase of testing is described by Ghiassi and Woldman (1994). It requires two programs to be written in different languages, where the program that will be delivered is written in a lower level language than the program that becomes the oracle.

The feasibility of this approach is based upon the concept that a higher level language program (HLLP) will take less time to write and be easier to test than a Lower Level Language Program (LLLP). This is because the HLLP will have fewer lines of code (LOC) than the LLLP. The authors suggest that, if the HLLP and LLLP have the same level of complexity and have been written by programmers with the same level of experience, then there is a direct correlation between the level of testing required and LOC. However, this statement does not stand up to investigation. LOC is the weakest complexity measure in common use. It does not correlate over different programming languages unlike the Cyclomatic Complexity Measure (CCM, see section 1.3.1). Watson and McCabe (1996) cite a number of case studies that show that errors in a program are more closely related to its CCM than to LOC.

Hoffman and Strooper (1991) describe the use of a Type IV oracle which has been written in Prolog. The Type IV oracle is a parallel implementation of the program using a higher level language (Prolog) than the release implementation (C). Hoffman and Strooper claim that the Prolog oracle cost little to implement.

However, the testing was at a low level (unit/module testing), so it is unclear how cost effective writing oracles to test larger systems would be.

Another experiment describes testing using M model programs (M-mp testing) (Manolache and Kourie 2001). This strategy requires M ($M \geq 1$) model programs to be written. Some of the criticisms earlier in this chapter, of the dual programming approach to testing are dealt with here. For example in M-mp, the model produced does not need to be as complicated as the program being tested. Instead the model(s) produced can be an abstraction of the program, perhaps a subset of the program's full features. When producing the model program, shortcuts can be applied. For example, in their experiment, Manolache and Kourie found that a code generator could be used to significantly shorten the time taken to produce the bulk of the code. They also suggest that the suitability of the M-mp approach depends upon whether it is more viable to produce and maintain model programs in terms of cost and time, compared to calculating the outcomes of tests manually. However, there is the outstanding problem, that bugs in the model programs are still likely to exist. This will require additional debugging of the model programs, and a mechanism to handle bugs found in them.

There are two ways that the model programs can be used. The first is to run the model program(s) independently from the program being tested (see Figure 2). The outputs from each program are then checked. This is using the model programs as Type IV oracles. The second is for the model programs to act as post-condition checkers with the output from the program under test being fed as inputs with the original input into the model program(s) (see Figure 3). In this case the model programs are being used as Type V oracles, where they check each x,y pair for correctness.

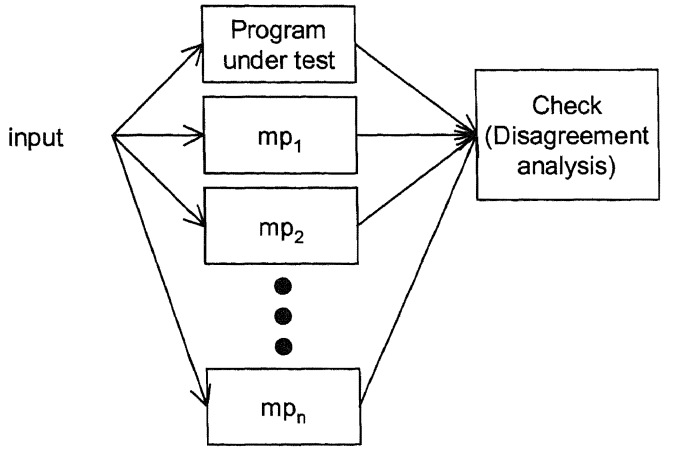


Figure 2 Model programs run independently of SUT (Manolache and Kourie 2001)

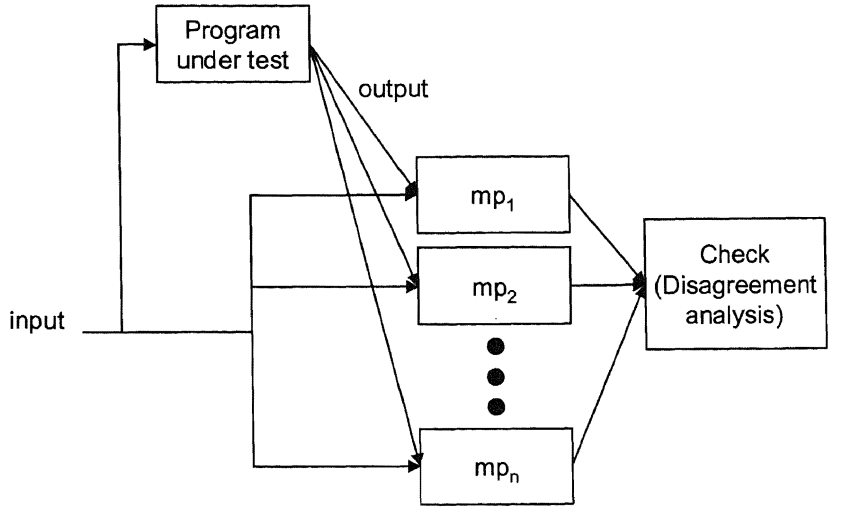


Figure 3 Model programs acting as Type V oracles (Manolache and Kourie 2001)

Edwards (2001) proposes a strategy for automated black-box testing of software components that uses oracles as pre-condition checkers (checking input conditions) and post-condition checkers (checking output conditions). The components being tested are object oriented classes. These classes contain "hooks" that allow built in test (BIT) capabilities to be added to the component. These capabilities include pre-

condition checking and post-condition checking. In effect the BIT capabilities wrap the component, leading to the term "BIT wrapper". The BIT wrapper is used for unit, component and integration testing. It is removed before system testing and release, as there is an impact upon the speed of the tested components. In effect the BIT wrapper is an extension of the common practice of software engineers to use assertions in code. Assertions are often used to confirm the values of input parameters into functions and they are usually only active in debug builds of code.

A further extension of BIT wrappers is contract-checking wrappers (Edwards *et al* 2004). A contract-wrapper can be applied to a class, either by the developer of the class or a client of the class. The contract-wrapper detects pre-condition and post-condition violations of the class. They are an improvement over assertions as they can be switched on or off at runtime (without recompilation being required), and can be used by component clients, not just component developers.

Relative Debugging

This section looks at a tool called GUARD (Griffith University Relative Debugger), which aids the debugging of evolving software (Abramson and Sosic 1996). It has been included in the literature review because Abramson and Sosic explored the concept of using previous versions of software to help the debugging process. Though this is not testing, this is the closest research found during the literature review to the concept of using software that already exists as an oracle in software testing.

According to Abramson and Sosic (1996) Software changes for a number of reasons. These changes can be classified into two types. They are migration and functional changes. Migration occurs where code is rewritten in a new language or moved to another hardware architecture or operating system. Functional changes occur where new features may be added, or algorithms changed to be more efficient. The use of traditional debuggers to locate areas in a program where a

more recent version of software has diverged from an old version is an error prone and tedious technique.

GUARD executes the program being debugged and a reference (oracle) program. It compares the contents of data structures in the two programs and highlights areas where there is a difference.

This differs from the other approaches discussed, in that GUARD looks at the internal differences between programs rather than the external differences. This can be related to mutation testing, where strong mutation testing requires a difference in the output and weak mutation testing requires a different internal state after a test has been executed.

GUARD works over a distributed network, allowing the programs to run on different machines and possibly different operating systems. This makes it necessary for GUARD to be able to deal with differences caused by different systems (e.g. floating point representation). GUARD uses a tolerance value, either globally or locally defined, relative or absolute. Numbers are defined to be equivalent if they are within this tolerance value.

GUARD makes no assumptions about the flow of control in the two programs. Instead, the user must determine key points in the program where various data structures should be equivalent. They then give GUARD a list of assertions that reference the program's variables and lines within the code. GUARD then uses these assertions to stop the executing programs and compare the specified variables. GUARD can then either highlight problems, or continue executing automatically if there is no difference.

These assertions can also be added to the program's comments. A pre-processor is then used to extract the "partial assertions" from the two programs and automatically generate the list of assertions, prior to debugging.

With the release of the Microsoft .NET framework a new version of GUARD has been implemented that is integrated into the MS Visual Studio .NET development environment. This has been used to show that GUARD can be used when porting applications from WIN32 to .NET and for cross-platform debugging between a UNIX platform and a Windows platform (Abramson *et al* 2002, Abramson and Watson 2003).

Abramson and Sosic (1996) have outlined the use of an oracle based approach to aid debugging. It is not a fully automated method, as it relies on user interaction to specify areas where the systems should be comparable, to execute the code and to restart the program when differences have been detected.

Prototypes

Prototypes are often developed prior to the final software being produced. These prototypes could be used to act as an oracle for module or unit testing. However, the prototype may produce output in a different form to the final system. Staknis (1990) proposes the use of an interface to overcome this problem. The automated testing process is as follows (see Figure 4):

1. The harness supplies test input to the interface.
2. The interface transforms the input into a form suitable for the module being tested.
3. The interface invokes the module with the transformed input.
4. The results are returned to the interface, which transforms these into a suitable form for the harness.
5. The harness compares the actual results against the expected results (generated by the prototype) and logs the results and test data for future investigation.

There are two options for generating the results. The first option is to use the prototype prior to testing of the module under test. The prototype is run to generate the results, which are saved in a file of input-result pairs. The harness then reads these input-result pairs. The second option is to run the prototype concurrently with the module being tested. In this case, Staknis suggests linking the prototype into the testing harness.

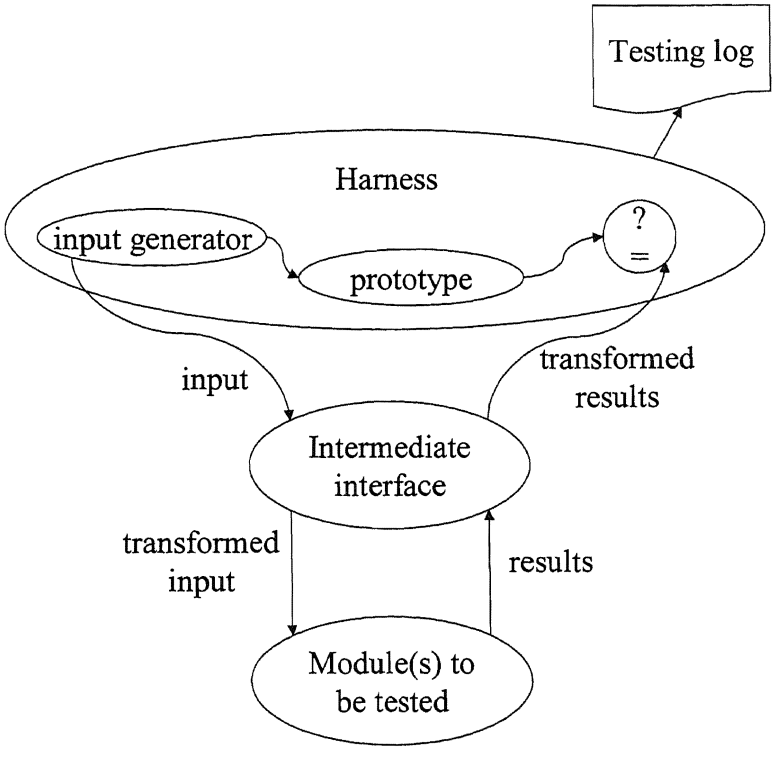


Figure 4 Structure for dynamic automated testing

There are problems with using prototypes as an oracle. A major problem is the availability of functional prototypes. Prototypes are often written as part of the rapid application development (RAD) process. These may have some functionality, but are likely to be quite limited. This makes such prototypes unsuitable to use as an oracle for testing. In practice, it will be very rare that a fully functional prototype will be written. If a functional prototype is written, it is likely to be used to establish the algorithm of part of the software being developed. When the final version is written the code from the prototype may be transferred over to the final

version with minor changes (if the programming language and platform are the same) or be re-written in a different language. Even if the algorithm is re-written any bugs in the prototype algorithm are likely to be transferred to the new version. It is likely, therefore, that there is a high probability of coincident errors between the module being tested and the prototype; certainly a much higher number of coincident failures could be expected than for programs developed independently as in *M*-mp testing. Therefore, it is unclear just how many bugs are likely to be found when using a prototype as the oracle.

Oracles from Formal Specifications

One solution to the oracle problem is to produce an executable oracle from the formal specification. This is the approach taken for the QUEST project (Brown *et al* 1992). The formal specification is in IORL (Input Output Requirements Language), a graphical formal language. From the specification two programs are produced. The first is manually coded, while the second is automatically generated from the specification. The automatic code generation is carried out with a "sim-compiler" ("sim" is short for simulation). The two independently produced programs can then be tested together and the results compared.

The sim-compiler produces code from the formal specification, that is neither as efficient nor as reliable as that produced manually. For this reason, the sim-compiler cannot be used to generate software for release. Brown et al (1992) also mention shortcomings in the sim-compiler that would be likely to cause inconsistencies between the target code and the oracle. During testing these would result in false negatives (test failures). False positives are less likely as the probability of the same error occurring in both the target and oracle is very small. These problems will disappear as the sim-compiler matures.

When discussing the choice for the formal language the authors note that there should be a meaningful difference between the design specification language and the language used for the target system. If not, this approach would result in

redundant coding, which would at least double the cost of the software development phase (a criticism of the n-version programming approach).

The use of a graphical formal language alleviates one criticism of using formal languages in commercial software development environments. A graphical representation of a problem is likely to be easier to learn and understand than a formal mathematical representation. However, the work by Brown *et al* (1992) did have one major flaw. The IORL specification produced was "reverse engineered" from the finished FORTRAN program. It is difficult to see how results claimed for a process which starts with the development of a formal specification, can be considered valid when the specification was actually generated from the finished program!

VDM (Vienna Development Method) specifications can also be used as automated oracles (Aichernig 1999). This method concentrates on using the post-conditions specified in VDM. The post-condition is the relation between the input, the old state, the output and the new state. The test input is passed to the function being tested. It is also passed to a pre-condition check, which validates the input, before being sent to the post-condition oracle. The output from the function being tested is passed to the post-condition oracle. The oracle checks to see whether the post-condition evaluates to true. Aichernig's approach concentrates on testing a single function. This makes it unclear whether it will be possible to scale up the approach to test the many functions and modules that are part of a normal software development project.

Abstract Data Type (ADT) formal specifications can also be used to form the basis of a test oracle (Antoy and Hamlet 2000). One difficulty in using a formal specification is relating the abstract representation (the formal specification) to the concrete code (the implemented program). There needs to be a transformation between the abstract and the concrete. Antoy and Hamlet propose writing a representation function relating the structures of the implementation to the abstract terms. Once the formal specification, implementation (a class in an object oriented

language such as C++) and the representation function are written, then testing can proceed using a standard unit test strategy.

There are two major drawbacks to using oracles based on formal specifications. The first is the lack of uptake of formal specifications in commercial software engineering environments. With the exception of the graphical specification (IORL), such specification languages require the use of very specialised skills. The second major drawback is that the use of formal specifications as oracles is limited to unit testing. It is very difficult to see how such methods can be expanded to module, integration or system testing.

Extracting Oracles from Documentation and Code

Brown *et al* (1992) reverse engineered their formal specification from the program FORTRAN code. This is one approach that can be taken when the specification does not exist for a program.

However, a high level specification should provide a view of the application domain (real world) and a description of what will eventually become a program which solves some problem in the application domain (Brown 1993). The specification provides a link between the program and the application domain. Reverse engineering produces a high level description of the code rather than modelling the application domain. In a poorly structured system it may be difficult to predict the effect of an operation, particularly where pointers or other such constructs are used. This means that it may be impossible to obtain the information required from the source code alone. In addition, though it is possible for small programs to develop specifications that completely define a program's actions, large systems require abstraction, otherwise the specification will become too large and cumbersome. Finally, there is evidence that the style of specification produced will affect the way that code is examined (Brown 1993). For example, an object-oriented approach will cause an engineer to look for possible objects and hierarchies

of objects, while a state-based specification style will result in the engineer looking for system states.

There is also a question of whether it is appropriate to use such specifications, or other data obtained only from the program code, directly for testing purposes. Details obtained from the code show what the code actually does, rather than what the code should do. Therefore, any oracle based on such a specification is a self-fulfilling prophecy.

There are ways to overcome this problem. The use of structured natural language within the code could be employed to help generate a useful specification (Curtis *et al* 1998). The details for this approach are covered in Chapter 2 section 2.2.1. Another approach is to use a system such as SIFT.

SIFT (specification information from text) parses program documentation, such as natural language specifications and end-user documentation (Lutsky 2000). The output from SIFT is information relevant to testing the application. SIFT consists of four modules: two are domain independent, and two are domain dependent. The independent modules are the "parser" and "testing knowledge". The domain dependent modules are the "sub-language grammar" and the "domain model". SIFT is used to augment the test cases that are manually developed; it does not provide a means of producing an oracle. SIFT relies upon a restricted language being used. If language styles vary greatly throughout documents then SIFT will struggle to extract information. In addition SIFT needs the documents to be structured clearly. For example each entity being tested needs to have a separate sub-section.

Currently there appears to be no work that uses program code and natural language documentation (or code comments) to produce an automated oracle.

1.3.4 Test Frameworks

There is very little research into the practicalities of managing and executing tests. However, details of one testing framework known as STAF (Software Testing Automation Framework) have been published (Rankin 2002). This section reviews STAF and other test frameworks that are available both commercially and as Open Source software.

First to be considered are regression testing frameworks such as WinRunner (Mercury 2005a). This is essentially a capture/replay tool. It allows tests to be recorded on the system, and played back for future regression testing. Scripts are generated that can be edited. "Checkpoints" can be added to the scripts that allow the comparison of expected and actual outcomes. The literature claims that WinRunner can compare data of different types, and check values against the SUT's database. The main problem with such tools is that they allow a suite of tests to be developed that are difficult to maintain. For example, if a small change is made to the GUI of the SUT, (e.g. the checkbox to set or clear an option is moved to a different dialog box) then all tests that use this option will need to be changed. It should be possible to modularise tests and create a library of functions, but this is then a serious development effort. In addition, it is easy to see that though testing may initially have been very thorough, the existence of the suite can allow complacency, and the set of tests can become stale and ineffective. Other similar products are QuickTest Professional (Mercury 2005b) and TestWorks CAPBAK (Software Research 2005).

SilkTest (Segue 2005) is another commercial testing tool. This is basically a sophisticated capture/replay tool. It uses a scripting language that models Graphical User Interfaces as objects, allowing for greater control when writing tests or maintaining the tests. Data driven testing is possible using a database to control the testing. Agents allow testing to be distributed, so that many machines can run tests while being controlled from a central test server. The criticisms of WinRunner also apply to SilkTest.

Another product is Rational Test Realtime (IBM 2005b). It includes tools to support test coverage analysis and tools to create stubs and test harnesses for unit testing. This is a potentially useful test product. However, developers can write stubs very quickly themselves as they develop software, and there is usually little need for these stubs to be kept or maintained. Therefore Realtime's benefit is that it may reduce the time developers take to do these tasks. Coverage analysis tools are valuable, as they provide quantitative data on how thorough the unit testing has been.

JUnit (JUnit 2004) is open source software. It supports the writing of test cases in Java, and mechanisms for running a set of tests together. The test cases require the inclusion of success criteria, though there are mechanisms for making writing similar tests easier. As with any automation effort, the tests resulting from this suite will need a great deal of maintenance.

Expect, DejaGnu and Tcl are all open source software that can be used to help with writing automated tests. Tcl (Tcl Developer Xchange 2005) is a general purpose scripting language. Expect (Libes 2005) is a specialised tool that enables scripts to be written that interact with a program, rather than running programs in batch mode. For example, Expect can be used to respond to a login, entering the username and password when prompted. This is not possible using standard shell scripts. Expect is written using Tcl. DejaGnu (Savoye 2005) is a testing framework based on Expect (and implicitly Tcl). It allows sets of tests to be written, and these can be executed on the system under test.

STAF is designed to solve the problems of reuse and automation. Rankin (2002) describes the design and development of STAF at IBM. A software testing framework was required that allowed reuse of libraries within tests. This reuse would allow test teams to use existing solutions rather than reinventing the wheel each time they needed to automate testing. Automation support was also needed that would allow the tests to be easily distributed to many client machines, executed on those machines and remotely monitored.

STAF is designed round a number of reusable components, called services. Each service provides a specialised set of functionality. Users can pick and choose which services they require, allowing them to only install the parts of STAF that they need.

At STAF's core are services that enable interprocess communication and queuing. Other services include: synchronisation using semaphores (both mutual exclusion and event semaphores are available), process execution and control, basic file system support, logging and remote monitoring. Support is also available that allows additional services to be added.

STAF has now been made available as an open-source product on SourceForge (STAF 2005). It is a framework that supports the manual development of sets of tests. It does not solve the problem of how to create and maintain tests or evaluate the outcome of tests.

To summarise, tools available commercially or as open source that support the automated execution of tests, revolve around three camps: Capture/Playback tools; frameworks to allow the execution of manually created tests; and tools to support the creation of stubs. None of these approaches allows the automated creation of tests. In addition, there is a problem maintaining these suites of tests. It is very easy to write (or record) a set of tests. However, when changes are made to the SUT, the suite of tests will need to be updated. This is particularly true of testing directed at the GUI, as minor changes to the layout or contents of a form, or the number of dialog boxes, will require the same changes to be made to the suite of tests. If the test suite is designed to be modular, then the changes needed may not be too drastic. However, this requires treating the automation of tests as a significant development exercise in its own right.

1.4 Implications of Literature Review on Research

The outcome of the literature review has many implications upon the research undertaken and presented in this thesis. The literature review has revealed that some areas in the published research on software testing are lacking and that further work is appropriate. These areas are: the oracle problem; the use of models to generate tests; the uneven and disconnected coverage given to different areas within software testing research; the issue of maintaining large suites of tests; and the lack of techniques which are shown to work on large systems (such as those developed commercially). There is also a lack of research into software testing techniques appropriate to support evolutionary development.

1.4.1 The Oracle Problem

An oracle provides the expected answer to the test. Automated testing without an oracle is limited to robustness testing (i.e. trying to make the software crash). A great deal of the research into software testing either assumes the presence of an oracle or does not address the problem at all. Yet, in order to automate testing an automated oracle is required.

The most promising approaches to the oracle problem are executable oracles. For example, oracles developed as part of N-Version programming or *M*-mp and back-to-back testing. The key advantage is that once the oracle has been written, a large number of tests can be run and evaluated quickly and efficiently. However, there are some disadvantages. For example, these oracles are written specifically to provide comparisons to the SUT and they have a high initial cost. Maintenance is also an issue. An oracle may be difficult to maintain and continue to use as a program develops over time.

The published research concentrates on using executable oracles for unit testing or module testing. Where testing did use a complete program (for example, Ghiassi and Woldman (1994)) the programs were small and trivial. It is not clear from this

research that an executable oracle could be applied to large software systems developed commercially.

Finally, there is the risk of coexistent errors, that is errors that are common to both the program being tested and the oracle. Ensuring that the oracle has been produced in a completely independent manner (using different design techniques, implementation language, etc.) reduces the risk of such errors, though it does not eliminate them.

Most of the work on using executable oracles assumed that the outputs from both the program being tested and the oracle were identical. Where they were not identical this was due to floating point inaccuracies over different programming languages or architectures. However, there are many potential sources of executable oracle, these include:

1. One derived from a formal specification.

2. A model of the software.

3. Specifically written software that models key behaviour but does not implement full functionality (heuristic oracle).

4. A previous version of the SUT (as in regression testing). This may be an internal release of the software, for example a version prior to a bug fix.

5. A previous release of the software being tested. This would be a release of the software that customers will see, i.e. an external release.

6. Software that performs (in part) a similar function.

7. A prototype.

8. A version of the software produced when multiple versions of the same application are written in an n-version programming or dual programming environment.

9. Similar software, possibly from a competitor.

10. A Reverse Engineered Specification (see Chapter 2, section 2.2.1).

The oracles of most interest to this research are "Type IV" oracles (Hoffman and Strooper 1991). Given any input, Type IV oracles can automatically generate the correct output (see page 32). However, as discussed on page 31, the oracle may not be perfect. All of the oracles listed above can be Type IV oracles (with the exception of the Reverse Engineered Specification which can be used as a Type III oracle).

1.4.2 Test Case Generation using Models

The automatic generation of test cases requires models (such as a Markov Chain Usage Model) to be built, or formal specifications (such as VDM or UML) to be written. The Markov Chain Transition Matrix is a powerful way of ensuring that the tests generated are meaningful. However, none of the work currently published on statistical testing with a Markov Chain Usage Model addresses how the Markov Chain Transition Matrix may be built automatically by a test system given a small amount of data.

Markov chains allow stochastic testing for statistical purposes. However, no work has been done on suitable oracles for use when testing with a Markov chain. The Markov chains also require a lot of work to build the usage model.

1.4.3 An Holistic Automated Testing Methodology

The published research takes a piecemeal approach to software testing. Even though to test software it is necessary to create tests, execute those tests and

establish whether the SUT has passed or failed the test, no research investigates how different techniques for these phases can be used together. Very little work has been carried out on the frameworks needed to manage tests. However, commercial and open source solutions are available.

The research presented in this thesis takes an holistic approach to automating software testing and tackles the whole process. For example, the oracle is predecessor software that may not be a perfect oracle. This imperfection is handled by the comparison phase. The comparison method directly influenced the form of the interface and the methods used to generate and execute the tests. The methodology developed also includes details on a framework to manage the automated testing.

1.4.4 Maintenance

The issue of maintenance is ignored. Automation of tests often results in suites of tests that are run against the SUT. When the SUT is modified (as it will be often during its development and subsequent life) the suite of tests must be maintained. The research presented in this thesis explicitly addresses the issue of maintenance.

1.4.5 Automated Software Testing for Industry

Research is dominated by techniques that are appropriate for small programs or for unit testing which will not scale to large systems. The reasons for this scalability problem are: the unacceptable effort needed to build models of large systems or to write suitable oracles; or that these techniques employ formal methods which are not extensively used in industry.

Companies in Europe and the US are moving away from using a "set in stone" specification and moving towards flexible project management techniques, as exemplified in evolutionary delivery and extreme programming. These techniques make many small incremental releases and obtain feedback from the customer

regularly throughout the process. As a consequence software testing should also be flexible to cope with the changing specifications.

Much of industry is using evolutionary delivery or iterative approaches to the development of their software. Yet the published research concentrates on approaches to software testing that are inflexible. They also ignore the large number of potential oracles that evolutionary delivery will produce.

Software testing is an important part of the development process that delivers good quality software. However, for maximum benefit it should be used in combination with other techniques (for example inspections) that require a large manual effort. However, project overruns and the need to meet deadlines often squeeze software testing. Therefore software testing should be as fully automated as possible. This means not just generating tests automatically and selecting the tests that may have greatest yield, but also looking at the whole process of running the tests, examining the outcomes of those tests and maintaining the tests.

This research addresses these problems by examining automated software testing of larger software systems, and using predecessor software as an oracle to the SUT.

1.4.6 Black Box or White Box?

There are many approaches that can be taken to automatically create tests. The first is to use a white-box approach and examine the source code of the SUT, using this to guide the testing. However, if this approach is taken then the tests created are potentially self-fulfilling. That is, the tests demonstrate that the code does what it says. The use of an oracle mitigates this somewhat, but the tests will still not detect code that is missing.

An alternative approach is to use the code of the oracle to guide the testing. This avoids the problems with self-fulfilling tests. However, the methodology developed uses software predecessors for the oracle. There is no guarantee that the oracle's

source code would be available and reverse engineering software owned by another company is unethical and potentially illegal.

Therefore, the approach taken is to treat the SUT and the oracle as black-boxes. This means that the tests generated are based on the functionality of the software, rather than on the software's internal code.

1.5 RESEARCH AIMS

The aims for this research are split into three parts. Firstly, there is a set of criteria by which the success of a testing methodology can be evaluated. Secondly, there are objectives that the methodology developed should meet. Finally, there are specific questions that need to be answered.

1.5.1 Criteria for a Successful Test Methodology

The following criteria for a successful test methodology have been developed as a result of reviewing the current literature and are based on professional experience of commercial software engineering.

A successful test methodology will try to:

1. minimise the manual effort needed.
2. maximise the likelihood of detecting an error.
3. minimise the likelihood of reporting false negatives.
4. minimise the likelihood of reporting false positives.

These criteria need explaining:

1. Minimising manual effort does not mean removing all manual effort. It is envisaged that some manual effort will be needed when testing software. However, the effort applied should be utilised properly. If manual effort is

applied, it should be clear where high skills levels are most beneficial, and where a low skill level is all that is required. This makes it possible in an industrial software development environment to allocate effort properly within a team of testers. All repetitious and long operations should be automated.

2. Error detection rates can be increased in two ways: firstly, by using techniques to target the locations of software where errors are most likely to occur (for example, testing at boundaries); and secondly, by using methods that allow a large number of tests to be run quickly and/or with minimum manual effort.

3. A false negative occurs when a test is reported as failed, when in fact it has succeeded. It takes time to check the test that has found a failure and establish that a failure has not actually occurred.

4. A false positive occurs when a test is reported as passed, when in fact it has failed. A false positive is a failure that should have been detected by the test system, but has been missed. A false positive will not be examined, allowing a bug to pass through when it should have been detected. A false positive is different from an error that has not been detected. An error remains undetected because a test has not been run that triggers the error and causes a failure in the software. A false positive occurs when the software behaves incorrectly, but this is not reported as a failure.

1.5.2 Objectives for the Methodology Developed

The methodology developed should be applicable for use in commercial software development environments. There are a number of common-sense objectives that need to be met. They are stated here for clarity, and are given in order of priority:

1. The methodology should result in test suites that are maintainable. Often regression test suites develop into a large number of unmanageable and unmaintainable scripts. Any changes in the software will result in scripts that are broken (they fail to run for some reason). It becomes difficult to see which

scripts are actually testing the software usefully, and which scripts do not exercise the software in a way that is likely to find any errors. Automation can result in an unmanageable set of scripts that when run are unlikely to find any bugs.

2. The methodology must be usable by software testers and developers.

3. A minimum of training should be necessary to be able to use the methodology in a productive way.

Testing can be carried out at throughout the software development life cycle. However, most techniques in the literature are aimed at the early phases (mostly unit testing). In this research a broader approach is taken and the techniques investigated are applicable to module, integration and system testing.

1.5.3 The Research Questions

The literature review highlighted important areas in software testing research that are poorly explored, or where more work is required (see section 1.4). Table 3 lists the key questions that need to be investigated in this research. These questions are the result of the conclusions drawn from the literature review.

The first question regards the oracle problem. For any automation effort to be successful an oracle is needed. The form of the oracle must reflect the target of this research, commercial software development environments. For example:

- Many organisations that are developing software commercially are not using specifications that are set in stone for the project. Gilb (1997b) states that you cannot know everything about the requirements, design, costs, timing and risks of a project. Therefore, flexible project management practices, such as evolutionary development and extreme programming are increasingly being used.

- Most software engineering projects do not use formal methods (Glass 2004). Therefore an approach that uses formal specifications is not applicable in many software engineering environments.

Research Area/Topic	Question number	Question
Oracle problem	1	What form should the oracle take?
Maintainability	2	How are tests written using the new methodology to be maintained?
	3	How are changes in the software reflected in the work required to maintain the test suite?
	4	How should "broken" tests be managed?
Automation	5	How much manual effort is needed to implement, run and analyse the results from the tests?
	6	What is the minimum manual effort that can be expected?
Procedures	7	What procedures should be followed when creating a test and managing the test throughout its lifetime?
Implementation	8	Is it better to write a test as a small executable program, compiled from code, or should a test case be written in an interpreted language?
	9	What are the advantages and disadvantages of using an interpreted language or a compiled language for implementing the tests?
	10	Are there benefits to combining interpreted and compiled languages when writing tests?
Management	11	How should tests be managed?
	12	What data are needed, and how are these to be managed?
	13	What about management of the same set of tests over multiple platforms?

Table 3 Research questions

Questions two to four relate to maintainability of the test created. Everyone can see what a test is intended to do while it is working. However, if a test "breaks" because the program has changed and can no longer be run, then it can be much more difficult to see what the test should be doing.

Questions five and six examine how much manual effort may still be needed for the automated methodology. Question seven addresses the procedures that should be

followed when creating and maintaining a test. The next questions relate to the choice of language technology that should be used to implement tests. Finally, questions eleven to thirteen examine how tests and test data should be managed.

1.6 Novel Features of the Research

The novel features of this research are:

1. A solution to the oracle problem is investigated. Predecessor software is used to provide an oracle. Predecessor software is software that exists prior to the testing of the system under test. This may include direct ancestors that are previous versions of the SUT or indirect ancestors that are software written independently from the SUT but perform (in part) similar functions.

2. The selected oracle will probably not be perfect. These imperfections need to be handled. A combination of an interface which describes the SUT as small units of functionality (UOFs), and the use of rules which are applied on the test output enable this imperfection to be handled.

3. A Markov Chain Transition Matrix (MCTM) is automatically created from the UOFs in the interface. The inputs and outputs for the UOFs are used as the basis for creating states in the MCTM. Algorithms to calculate the states and build the MCTM are presented. Another algorithm to check that the MCTM is consistent is also presented. These algorithms have not been published previously.

4. An holistic methodology has been developed. It addresses all phases of the process of automated software testing from test generation through to evaluation of the test results.

5. The issue of maintenance when developing automated test suites is explicitly addressed.

6. Commercial software development has been considered. This means that the methodology developed concentrates on testing of larger pieces of software (such as component, sub-system, integration and system testing). Prototype software (called Alltest) has been developed with the goal of being practical and applicable to real world software development. Alltest has been used to test real commercial software that is sold and in active use.

1.7 THESIS STRUCTURE

This chapter has reviewed the current literature relating to software testing and the quality of software produced in commercial environments. It has drawn some conclusions from the literature review and identified key questions that need to be addressed by this research.

The next chapter describes the methodology developed during this research and the design and implementation of Alltest, which was written to implement the methodology. It also explains the novel Markov Chain Transition Matrix generation algorithm. Chapter 3 describes how this research has been evaluated, the outcome of the experiments and finishes with a discussion of what the results mean. Finally, Chapter 4 reviews the contribution to knowledge made by this research. It finishes with a discussion of future work resulting from this research.

CHAPTER 2 AUTOMATED TESTING METHODOLOGY

When testing a software product there are three distinct phases, generation of tests, execution of tests and evaluation of results. Of the three phases, evaluation of the results is the hardest to automate. To carry out evaluation of the test results by hand is slow and tedious. Therefore, if generation and execution of the tests are automated, it is essential to automate the evaluation phase, as much as possible, to avoid a bottleneck in the testing process.

Whatever techniques are used for testing, there are a set of processes and outputs that can be clearly defined. Figure 5 shows a process flow chart that defines these different stages. It shows all the processes that must be carried out and all the outputs (documents) produced. It also shows two databases. The first database *(E)* contains the test data and expected results. It also contains the actual results obtained from running the tests. The second database *(F)* contains the list of bugs (or suspected bugs) found during testing.

The three phases of testing can be seen in Figure 5. The tests are generated *(5)*. They are executed *(3)* and their results are evaluated *(4)*. Also shown in this diagram is how testing fits into a general development process, starting with the specification *(A)* which is turned into code *(B)* by a development team. In a parallel process, the test cases are produced from the specification *(A →5 →E)*. Once the test cases exist *(E)* and an executable program exists *(C)*, the program can be tested *(C →3 →D →4 →E)*.

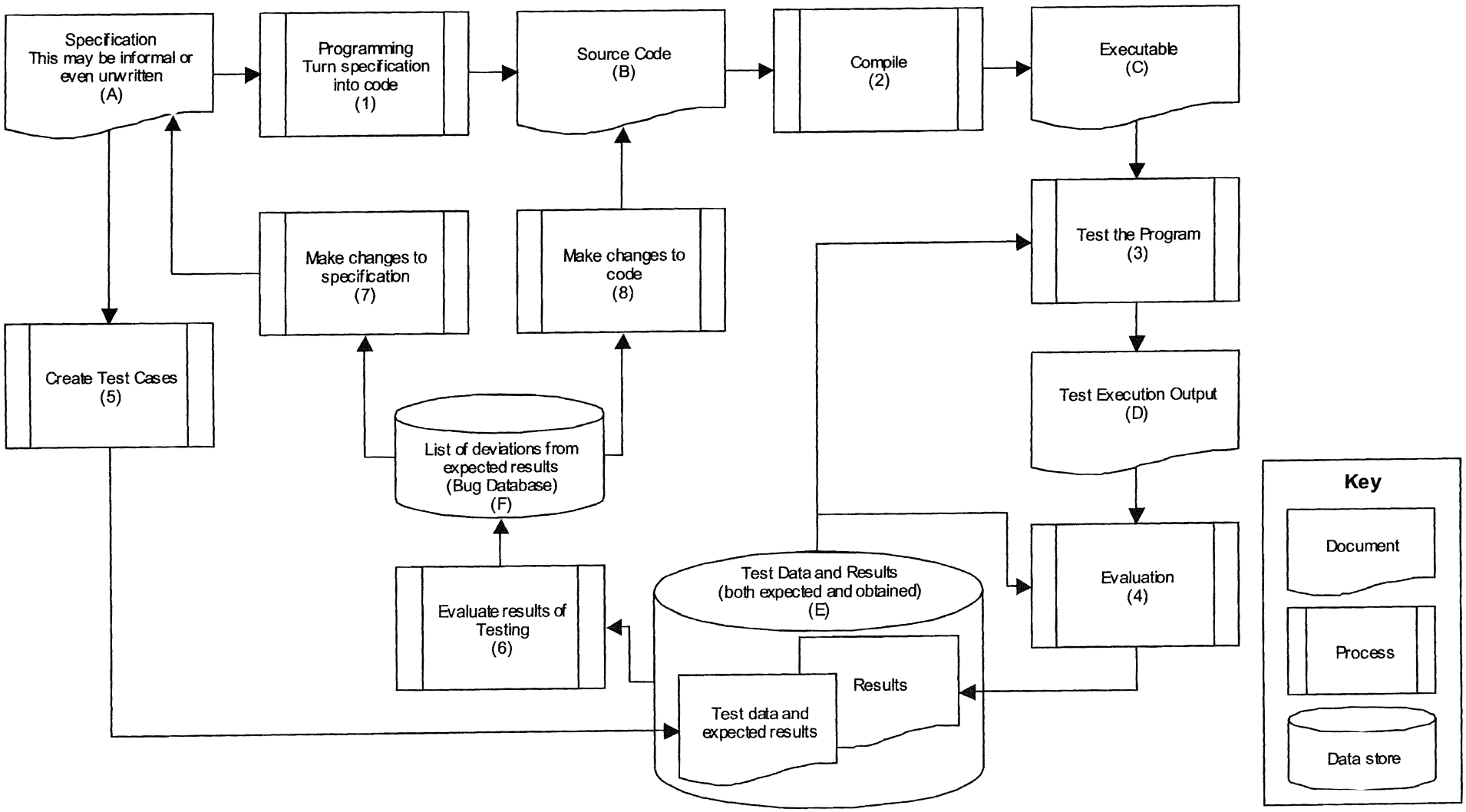


Figure 5 Test process workflow diagram

Following testing, the results are analysed and a list of defects is generated ***(E → 6 → F)***. This defect list will correspond to issues with the code or issues with the original specification. If there are issues with the code, then corrections are made to the code ***(F → 8 → B)***. Alternatively, any issues with the specification result in changes being made to the specification ***(F → 7 → A)***. This evaluation forms a feedback loop. Further tests may be generated from the updated specification, and the process repeated until some criteria are met that indicates that testing is "complete".

Figure 5 does not differentiate between automated strategies or manual strategies. Any of the processes ***(1, 2, ... 8)*** could be carried out manually or automatically. For the purposes of testing, it is usual to find that ***(3)***, executing the tests is carried out automatically. However, generation of the tests ***(5)*** and evaluation of their results ***(4)*** could also be carried out automatically. The methodology developed automates evaluation ***(4)*** by using an 'oracle'. This oracle provides the answer to the question "what is the expected result of this test?" This could be answered by examining a program specification or asking a computer operator. However, for the evaluation phase to be automated, an oracle that is automated (or can be automatically processed) is required.

Alltest is prototype software that implements the methodology. It consists of a test generator, a test executor and a test comparator, see Figure 6. An interface is also shown in the figure. This interface is developed for each application tested (and is not actually part of Alltest itself). Also on the diagram are the System Under Test (SUT) and the oracle system. Tests are executed on both these systems via the interface.

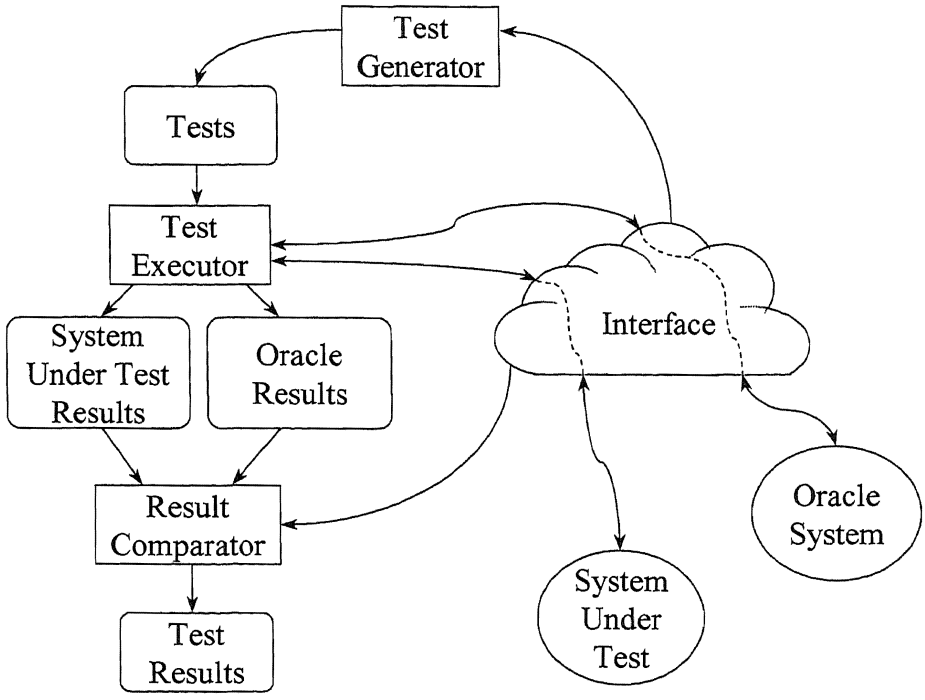


Figure 6 Automated testing with an oracle system

The rest of this chapter describes Alltest. It explains: how Alltest is implemented and how it is used; the novel automatic creation of a Markov Chain Transition Matrix to generate tests; and the novel use of predecessor software to act as an oracle for the testing process. Section 2.1 describes the interface. Section 2.2 explains what can be used as an oracle. Section 2.3 looks at the generation of tests and the creation of a Markov Chain Transition Matrix. Section 2.4 discusses how the tests are executed and section 2.5 examines the evaluation phase. Section 2.6 describes how Alltest is implemented and section 2.7 assesses the options available to manage the test process and the data used and generated when testing. Finally, Section 2.8 discusses how Alltest is used.

2.1 THE INTERFACE

The interface is not part of Alltest. It is written for each software system or component that needs to be tested. It describes the SUT in a way that allows tests to

be automatically generated. The interface enables the tests to be run on the SUT and the oracle system. The interface specifies how the results from testing are to be automatically evaluated. This section describes the interface and the files that implement it.

2.1.1 Unit of Functionality (UOF)

Functionality of the SUT is described as small units. Each of these units is called a Unit of Functionality (UOF). A UOF does not specify any behaviour of the SUT, but is a named black-box with inputs and outputs. The oracle specifies the behaviour of each UOF. As an example, if a text editor is being tested, then opening a file is one UOF, and saving a new file is another UOF.

2.1.2 TCD files

Each UOF is coded in a Test Command Description (TCD) file. Figure 7 shows an example TCD file. The TCD file consists of three sections. These sections are delimited with '%%'. The first section specifies parameters to the UOF. These are used by the generator when creating the tests. The second section specifies how the comparison will be carried out. The final section specifies how the test is executed.

2.1.3 Rules

Rules control how the comparison phase will be carried out. Rules can be defined within the TCD file or as separate script files. Rules are fully explained in Section 2.5.

2.1.4 Test configuration

The Test Configuration file contains information that tells the generator how to create the tests. For example it specifies how many tests should be created. The configuration file also includes parameters that are used directly by the generation

algorithm. For example, one parameter is the probability that the test will end (which controls the length of the generated tests).

```
# Copy a file
FileName1 -l255 -di string
FileName2 -l255 -di string

%%

# Second section: rules for the comparison
# Specifies global, and shared local rules
# Also defines new rules that are only used for this command

USERULE LINE SUB_PATHS
USERULE LINE NOCASE
USERULE FILE COPY_MULTILINE_FAILURE
USERULE LINE FS_FAILURE_MESSAGE

%%

# The interpreter that will be used:
%INTERP TCL

# copy the file (full paths will be supplied)

puts "Copying from [file nativename $FileName1] to [file nativename $FileName2]"

exec cmd.exe /c copy [file nativename $FileName1] [file nativename $FileName2]
```

Figure 7 Example TCD file for copying a file

2.1.5 Creating the Interface

Figure 8 is a flowchart of the steps needed to develop an interface. The first step is to look for a suitable oracle. The oracle is used when evaluating the output from running the tests. The oracle should be executable software that performs (in part) a similar function. This may be a previous version of the system under test, a prototype or similar software possibly from a competitor. It is important to remember that the oracle will not be perfect. This imperfection is handled by developing the interface.

Once the oracle has been selected, the SUT is examined and its behaviour is broken down into UOFs. For example, if a text editor is being tested, then opening a file is one UOF, and saving a new file is another UOF.

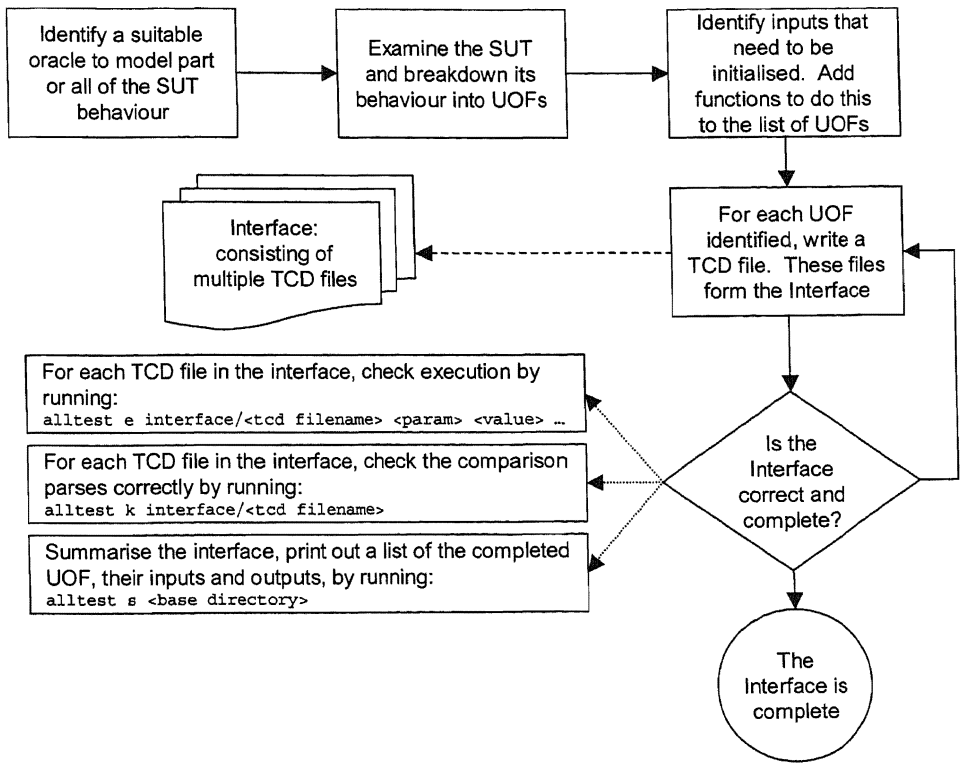


Figure 8 Developing the Alltest interface

The next step is to identify the inputs to each identified UOF. Continuing the example, a file location and a filename are needed in order to open a file. It will also be necessary to choose if the file is to be opened for editing or in read-only mode. In total these three inputs result in three additional UOFs for opening a file. For saving a new file there are again three inputs: file location, filename and if an existing file can be overwritten. However, two of these inputs are the same as for opening a file. Inputs only need to be specified once. So saving a new file only gives rise to one additional UOF for its inputs (see summary of UOFs in Table 4)

Function	Inputs	Additional Units of Functionality
Open a file	file path filename open read only	4 new UOFs: 1 action 3 UOFs that set parameters
Save a new file	file path filename overwrite existing	2 new UOFs: 1 action 1 UOF to set parameter

Table 4 Example Units of Functionality

For each UOF identified a TCD file is written. When writing a TCD file for an input it is important to consider both valid and invalid inputs. For example, when writing the TCD file to provide the file path, invalid characters should be considered. It is also important to "randomise" the file path returned. However "random" must be repeatable, so if a test is re-run, the same output is always produced. This ensures that the automated testing is repeatable.

Writing the TCD files builds up the interface to the SUT and oracle. The finished TCD files will specify their inputs and outputs, the rules used when comparing test output and the code to execute when running the tests.

Each TCD file written must be unit tested. Alltest has options to allow the execution and comparison of a single TCD file. This enables a TCD file to be run on the oracle system then on the SUT. The outputs from these can be examined to ensure they are as expected. These outputs can then be used for the comparison phase to ensure that the comparison is working. This unit testing minimises the number of false negatives found by Alltest when running generated tests.

However, there will still be occasions when the TCD files are incorrect (particularly the rules applied during the comparison phase). Therefore the next step is to carry out a few short test runs (perhaps with about 10 tests being generated) and analyse the results. Once any necessary corrections have been made to the TCD files full testing can commence.

2.2 THE ORACLE

The oracle provides the expected outcome from running a test. Alltest uses an oracle that can be executed. If the oracle is perfect then the output from the SUT can be compared directly against that from the oracle. If there are any differences, an error has been detected in the SUT. If there are no differences the SUT passed the test. However, the oracle used for Alltest is not expected to be perfect. See section 2.5 for details on how the imperfection is handled.

Some oracles can be treated as "predecessors" and these are discussed in section 2.2.2. However, we start by looking at the Reverse Engineered Specification in section 2.2.1.

2.2.1 Reverse Engineered Specification

Where there is only an inadequate specification available for a software product, one solution may be to use the code and its comments to provide the description of the software. Reverse engineering is used to produce this specification, which is called a Reverse Engineered Specification (RES). The author originally presented this idea at Software Quality Week, San Francisco, 1998 (Curtis *et al* 1998).

Figure 9 shows the steps taken to produce a RES. Starting with the source code, a computer program transforms the code into a RES, which uses a formalised natural language, based upon the ideas of Planguage (Gilb 1996, 1997a, 1997b) and graphical descriptions of the program. This RES is readable by humans and can be processed by computers.

The next stage is inspection, comparing the RES with existing specifications and/or expert knowledge. People possessing expert knowledge will include software engineers who have worked on the project, customers who defined the requirements, and people who have worked with software that performs a similar task. From this inspection a new "corrected" RES is produced.

At this stage problems located in the generated RES can be corrected by making changes to the original source code.

The RES will still be in the formalised natural language, and as such can be processed by another computer program to produce a set of test cases. The test cases will include input data and expected outputs. Manually produced test cases can be added to the automatically generated ones thus creating a suite of tests.

There are some problems associated with using Reverse Engineering techniques as part of the testing process. However, the process used to produce the RES helps to overcome these problems.

The first problem is that reverse engineering produces a high level description of the code that does not model the application domain (Brown 1993). Without modelling the application domain it is very difficult to comprehend what problem the software is trying to solve. The link from the application domain to the RES can be re-established by examining the comments included in the source code along with the program's structure. This ensures that all available information is being utilised.

The second problem when testing is that it is inappropriate to use data derived directly from the program code. This is because the details obtained show what the code actually does, rather than what the code should do (Beizer 1997). This problem is solved by including an inspection phase which feeds back into the creation of the RES and finally into the creation of the tests.

Finally, reverse engineering produces specifications that are very formalised. They are difficult to comprehend and restrict themselves to the analysis of program structure and algorithms. The result is an abstraction that lacks clarity. However, the formalisation is necessary to ensure that the resulting specification is not ambiguous, contradictory or incomplete.

Natural languages (such as English) can be used in a way that is ambiguous. Formal languages are precise, but are difficult to use without specialised training. One solution is to use a formalised natural language. This will resolve problems associated with both formal specification languages and natural languages. This forms a compromise that allows humans to easily read and modify the language, but also makes the language interpretable by a computer program. Planguage (Gilb 1996, 1997a, 1997b) has been chosen as the basis for such a language. Planguage is designed to formalise the specifications written by humans, it provides a way to

produce a specification that is easy to read by humans, yet avoids the problems of ambiguity often associated with natural language.

The use of Planguage also solves the problem of introducing information into the RES that is the result of examining comments in code. In addition, control flow diagrams and call graphs should be included as people often handle graphical information better than textual information.

Finally, information should be included in the RES that can highlight potential problems in the code, for example, the index for an array being used in a manner that allows a buffer overflow or underflow to be possible.

Figure 10 shows a RES being used as part of the testing process. This process is very similar to that shown in Figure 5 (see page 60).

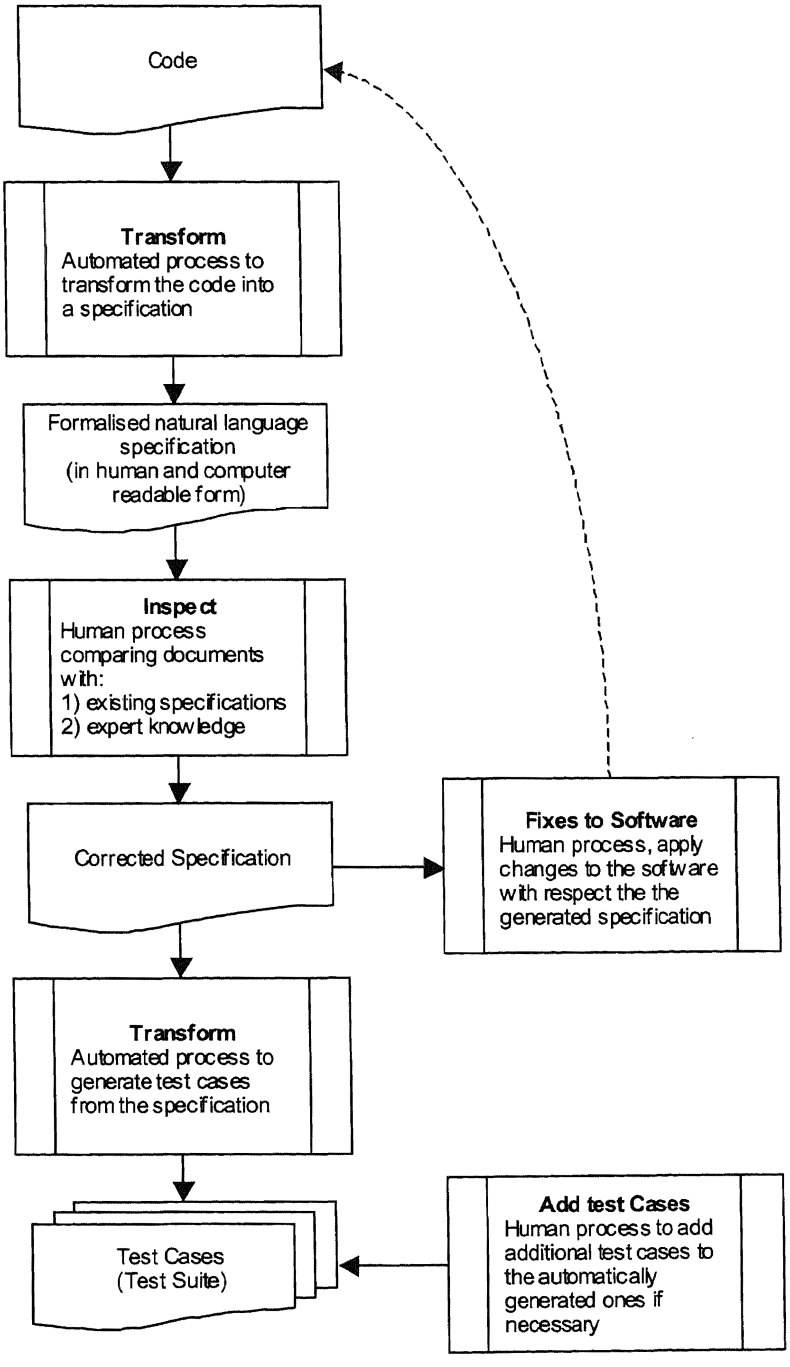


Figure 9 Test suite generation from code

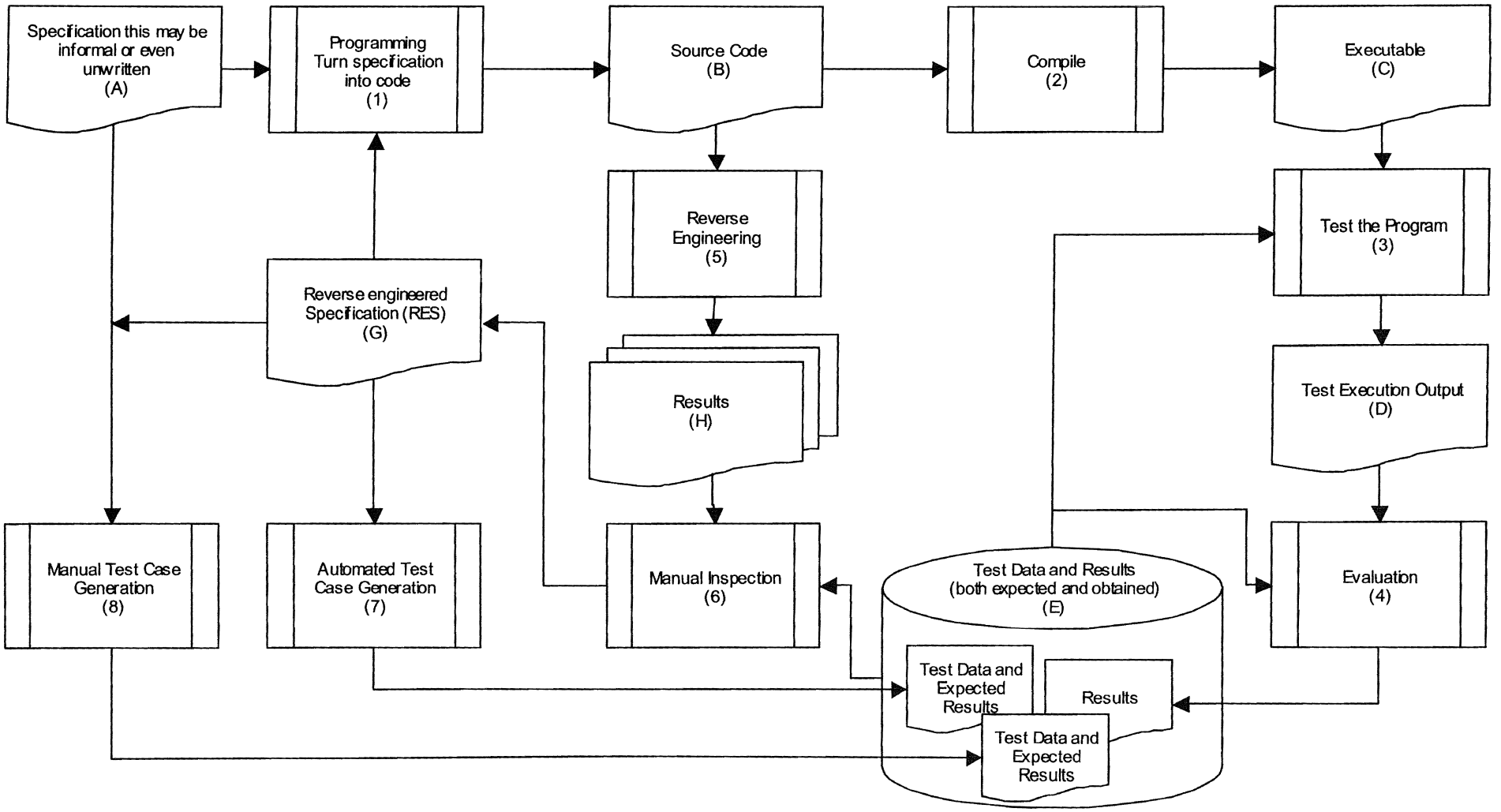


Figure 10 Workflow diagram for automated testing using a RES

2.2.2 Software Predecessors

Many of the oracles mentioned in Chapter 1 (section 1.4.1) can be considered to be predecessors to the SUT:

- Previous version – pre bug fix (as in regression testing).
- A previous release of the software being tested.
- Software that performs (in part) a similar function. For example, a file system performs a subset of operations carried out by Manager.
- A prototype.
- A version of software produced when multiple versions of the same application are written in an n-version programming or dual programming environment.
- Similar software, possibly from a competitor.

Predecessors are not limited to direct ancestors of the SUT with a common code base. For example, take a word processor in current use, such as Microsoft Word 2000. Its predecessors include Word 97, Word 95, Word 2, WordPerfect 5, Microsoft Works (which includes word processing abilities), WordPad and text editors. Alternatively consider a current operating system that would run the word processor, such as Microsoft Windows XP. Its predecessors include Windows 2000, NT 4.0, NT 3.51, 98, 95, Windows 3.1, 3.0, MS-Dos 6.0, 5.0, OS/2, Mac OS, Unix and VMS. Both of these examples show just some of the potential predecessors available to use when testing either application.

If a word processor is being tested, then WordPad could be used as a predecessor to test the formatting of text, for example, selecting text, making it bold, deleting text, applying italics and changing font. All of these things can be done with most word processors. However, these tasks may be carried out in different ways depending

upon the word processor, and the results may be displayed in a different manner (for example, not all word processors are WYSIWYG[1]). When using a predecessor as the oracle, these differences need to be managed. Section 2.4 describes how Alltest manages differences between the oracle and the SUT.

2.3 The Generation Phase

During the initial phases of this research, the test cases were created manually. This was because the aim was to concentrate on how the tests are evaluated. However, creating tests manually has many drawbacks. Tests written may be very similar, with only minor changes between each test. This makes writing the tests tedious and results in mistakes being made. The second problem is that even though the tests may be similar, it takes too long to write them manually. The third problem is one of maintenance. Once a suite of static tests is written, they need to be maintained, and that can take a great deal of effort. For these reasons the tests cases need to be automatically generated.

2.3.1 Stochastic Test Case Generation

As mentioned in Chapter 1 (section 1.4.6) a black-box testing approach has been adopted. Selecting a black-box approach to testing has ruled out the use of techniques, such as branch coverage, which use the source code to guide case creation. Instead, a stochastic test generation method has been developed. One reason for choosing stochastic test generation is that the users of the SUT can carry out operations is any (random) order, so the test generation should reflect this. Another reason is that the cost of writing a complex generation algorithm is high when compared against the cost of executing the tests and comparing their results. As both of those phases are automated, the cost of running and checking a large number of random tests is minimised. If the comparison phase had been manual, then using a random test generation method may have been inappropriate.

---

[1] What You See is What You Get

However, the output from the generator cannot be completely random. The generator needs to handle the order in which each UOF is called. It is necessary to ensure that input parameters to the UOF are initialised before the UOF is executed. For example, a browser cannot load a Web page until a URL has been provided. An approach is needed that allows the generator to know what functions are valid to be called. An algorithm that produces a Markov Chain Transition Matrix has been developed to deal with this.

2.3.2 Markov Chain Transition Matrix

As discussed in Chapter 1 (section 1.3.2) various practitioners have successfully used Markov chains to generate test data. Using a Markov chain allows the generator to keep track of which TCD files are valid to be called. This is done by linking variables (or combinations of variables) from the TCD files with the states in the Markov Chain Transition Matrix. The test is in a particular state when all the variables that describe that state are set. Additional states are the "initial" and the absorbing[1] "end test". The elements in the matrix contain a probability of making a transition from the current state to another state. An example Markov transition matrix is shown in Figure 11, with the initial state indicated as "0", and the end test state as "END".

State	Description	0	A	B	AB	END
0	No variable set (initial state)	0.3	0.3	0.3	0	0.1
A	variable A set	0.3	0.3	0	0.3	0.1
B	variable B set	0.3	0	0.3	0.3	0.1
AB	both A and B set	0	0.3	0.3	0.3	0.1
END	Absorbing end state	0	0	0	0	1

Figure 11 Markov transition matrix

By controlling the probability that a transition from one state to another will occur, Markov chains become a very powerful way of controlling the flavour of the tests generated. For example, tests can be made to be short or long. Repetitive tests can

[1] Parzen (Parzen 1960) defines the absorbing state as follows: "A state j in a Markov chain is said to be *absorbing* if $P(j,i) = 0$ for all states $i \neq j$, so that it is impossible to leave an absorbing state. Equivalently, a state j is absorbing if $P(j,j) = 1$."

be produced by carrying out operations on a single object multiple times, without changing the state. Alternatively, more varied and fluctuating tests can be produced where states are more transient. Alltest controls this with a parameter in the test configuration file: probability of staying in the same state.

At each state in the matrix there are also "functions" that can be called. Many of these functions will cause variables to be set (or cleared) and a transition into another state will take place. Some of these functions are from the executable part of the TCD files and fall into two types:

1. Functions that do not cause a transition between states. See Figure 12.

2. Functions that can be called to transition between the states. See Figure 13.

A third type of function is a notional function that is used by the generator to indicate that a variable in "unset". This "unset" function is the result of pairing functions that set the value of a parameter. The "unset" function clears the variable and allows the test to move into a previous state. Figure 14 shows a transition matrix with "unset" functions.

These three types of functions result in a complete matrix as shown in Figure 15.

State	0	A	B	AB	END
0	0	0.45	0.45	0	0.1
A	0.3	0.3 function3	0	0.3	0.1
B	0.3	0	0.3 function1 function2	0.3	0.1
AB	0	0.3	0.3	0.3 function4	0.1
END	0	0	0	0	1

Figure 12 Markov transition matrix showing same state functions

State	0	A	B	AB	END
0	0	0.45 set_A setInvalid_A	0.45 set_B	0	0.1
A	0.3	0.3	0	0.3 set_B	0.1
B	0.3	0	0.3	0.3 set_A setInvalid_A	0.1
AB	0	0.3	0.3	0.3	0.1
END	0	0	0	0	1

Figure 13 Markov transition matrix showing transition functions

State	0	A	B	AB	END
0	0	0.45	0.45	0	0.1
A	0.3 unset_A	0.3	0	0.3	0.1
B	0.3 unset_B	0	0.3	0.3	0.1
AB	0	0.3 unset_B	0.3 unset_A	0.3	0.1
END	0	0	0	0	1

Figure 14 Markov transition matrix showing "unset" transition functions

State	0	A	B	AB	END
0	0	0.45 set_A setInvalid_A	0.45 set_B	0	0.1
A	0.3 unset_A	0.3 function3	0	0.3 set_B	0.1
B	0.3 unset_B	0	0.3 function1 function2	0.3 set_A setInvalid_A	0.1
AB	0	0.3 unset_B	0.3 unset_A	0.3 function4	0.1
END	0	0	0	0	1

Figure 15 Markov transition matrix showing probabilities, states and functions

Figure 16 shows the inputs and outputs of the generator. The generator takes two types of file as inputs: multiple TCD files and a single Test Configuration file. Figure 7 on page 64 shows a TCD file consisting of three parts. The generator only uses the first part of the TCD file, which describes the inputs and the outputs to the UOF. Each line defines a "parameter". It specifies the parameter's type, whether it is an input, an output, or both, and any limitations on the values used. With these parameters defining the inputs and outputs, the UOF can be treated as a function in a procedural programming language.

Figure 17 shows a typical parameter definition. The first item is the parameter's name (in this case FooBar). This is followed by options that specify attributes of the parameter (see Table 5). FooBar is a string of not more than 255 characters, which is an input parameter to the TCD file.

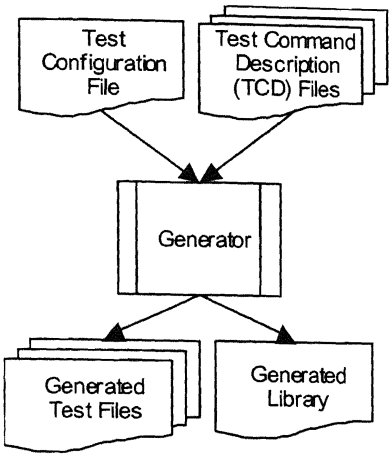


Figure 16 Inputs and outputs of the generator

```
FooBar -l255 -di string
```

Figure 17 A typical parameter definition in a TCD file

Attribute	Description	
`-ln`	Length	
`-rln` `-run`	Range	(lower) (upper)

`-di`
`-do`
`-dio`
	Direction	(input)
		(output)
		(input and output)

`string, bool, special, system, integer, float`	The final item defines the type of the parameter	

Table 5 Parameter attributes in a TCD file

Figure 18 shows the process carried out by the generator. The following steps describe how the test suite generates tests from the TCD files:

1. The TCD files are parsed, and a "function table" is generated. This table contains the information needed (such as function parameters) to enable each function to be used.

2. The Markov transition matrix is built. This consists of building a state table, which is then used to create the transition matrix. Building the state table is described in section 2.3.3.

3. A library file is created from the function table. This contains procedures that allow the executable part of individual TCD files to be run. The result is a one to one mapping between the library functions (in the chosen scripting language) and those functions described by the TCD files. In addition, the library file contains supporting procedures.

4. The tests are generated by taking a random walk though the transition matrix. The walk starts in the "initial" state. This assumes that no variables are set. A walk ends when it reaches the "end test" state. A walk between the "initial" and the "end test" states is a single test. The functions called are written to the test file generated. The test files are linear scripts that make calls into the TCD files.

5. Generation is complete. A set of tests has been created ready to be run on the SUT and the oracle system.

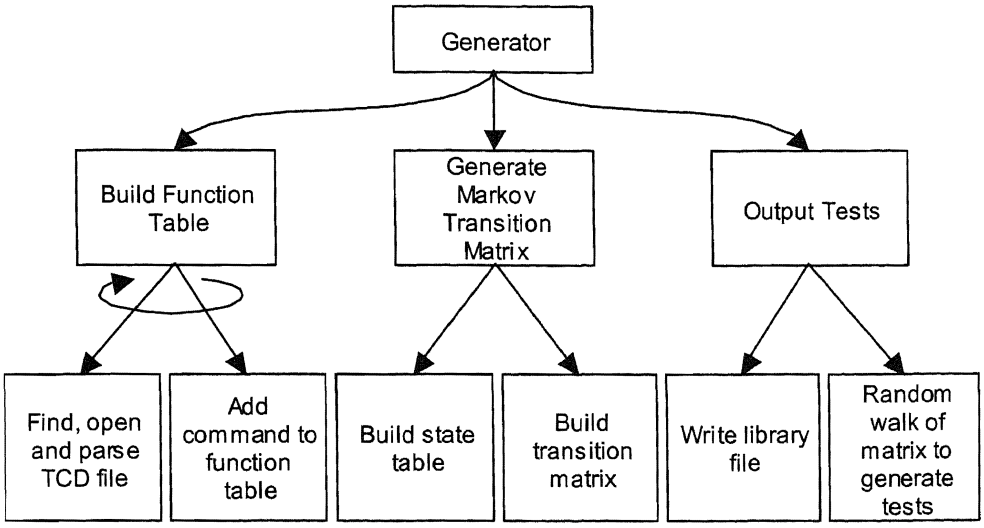


Figure 18 Breakdown of the generator

2.3.3 Calculating the States

At first calculating the states for the transition matrix appears trivial. However, it is a crucial part of building the transition matrix. If this stage is completed poorly then the tests generated may not include all functions that can be called. Alternatively the resulting matrix could be inefficient with unnecessary states that increase the length of tests without adding functionality to the tests. Worst of all, so many states may be created, that the resulting transition matrix is huge. For example, if the tests include a total of just 20 parameters, and the algorithm blindly generates all combinations, the total number of states is:

$$2^n - 1 = 2^{20} - 1 = 1{,}048{,}575$$

If these states are used to create the matrix, it will have the following number of elements:

$$1048575^2 = 1{,}099{,}509{,}530{,}625 \approx 10^{12}$$

If each element in the matrix requires just 4 bytes (a considerable underestimate) the resulting memory usage for such an algorithm would be in the order of 4 terra bytes. This can be mitigated by choosing a data structure that allows the matrix to be represented in a sparse fashion, with only the elements that contain functions actually having memory allocated. However, such an approach is still wasteful, with excessively long tests being produced, or the generator taking too long to walk the matrix to create the tests.

It is possible to reduce the number of states needed to a sensible minimum. This can be done by combining variables and treating these combinations as single entities. The variables are combined by examining how functions set or use them. The matrix produced must contain states that allow all functions to be called (no missing functions) and all states must be attainable (i.e. it must be possible to walk to all the states in the Matrix via the other states of the Matrix). By developing an algorithm to do this the Markov Chain Transition Matrix can be built automatically. Different algorithms were tried, some of these are described in Appendix D. An algorithm that checks the matrix generated has also been developed. This ensures that every state within the Transition Matrix can actually be reached. It is described in Appendix E.

Two of the algorithms are described below. Algorithm A has been used to generate tests producing the results described in this thesis. However, upon implementing the checking algorithm into Alltest, it became clear that all states in the matrix could not be reached, though all the functions were being called. A second algorithm (B) has been implemented into Alltest and used to generate tests. It produces a matrix with no states that are unreachable. The result of these algorithms is a state table that is used when creating the Markov Chain Transition Matrix.

Algorithm A

1. Go through the functions and build a Variable Table (VT). The VT contains a list of all output parameters and it is indexed on the output parameters. For each

function create a list of input and output parameters (in/out is treated as an input). Each output has an entry in the variable table that is expanded with the whole group of parameters from the function. For example if we have 4 functions with the following parameters $A^{(out)}$, $A^{(in/out)}B^{(out)}$, $B^{(in)}C^{(out)}$, $C^{(in)}$. Then the VT will be: A = {A}; B = {AB}; C= {BC}.

2. For each of the functions, create a list of all parameters.

3. Create combinations of these parameters for each function. For example Func(A, B, C) will have the following state combinations: A, B, AB, C, AC, BC, ABC.

4. Use VT to expand the combinations. For example A=A, B=AB, AB=AB, C=BC, AC=ABC, BC=ABC, BC=ABC. Removing duplicates gives: A, AB, BC, ABC.

Building the transition matrix can be illustrated by using five example functions:

```
One (out X, inout Y)
Two (in X, out Z)
Three (out Y, out W)
Four (in W)
Five (in Y, in Z)
```

Step 1, build the VT:

```
W=WY, X=XY, Y=WY, Z=XZ
```

Step 2, list the parameters for each function:

```
XY, XZ, YW, W, YZ
```

Step 3, enumerate the combinations

```
XY = {X, Y, XY}
XZ = {X, Z, XZ}
YW = {Y, W, YW}
```

```
W = {W}
YZ = {Y, Z, YZ}
```

Summarising:

```
X, Y, XY, Z, XZ, W, WY, YZ
```

Step 4, expand with the values from the VT.

```
X = XY
Y = WY
XY = XYWY = WXY
Z = XZ
XZ = XYXZ = XYZ
W = WY
WY = WYWY = WY
YZ = WYXZ = WXYX
```

Summarising this gives the states:

```
XY, WY, WXY, XZ, XYZ, WXYX
```

This produces the following matrix:

	0	XY	WY	WXY	XZ	XYZ	WXYZ
0			Three				
XY		One		Three		Two	
WY	Unset		Three Four	One			
WXY		Unset	Unset	Three One Four			Two
XZ					Two		Three
XYZ		Unset				One Two Five	Three
WXYZ				Unset	Unset	Unset	One Two Three Four Five

There are states available to call each function. Walking through the matrix using the checking algorithm (see Appendix E) shows that every state can be achieved.

Algorithm B

This algorithm produces a useable matrix with the test input files described in Chapter 3, section 3.3. It has been implemented, but its effectiveness for testing has not been evaluated.

1. Build the variable tree (VT). The variable tree is indexed using the variable names. For each function, create a list of output parameters (in/out is treated as an output) and add these to the variable list. The in/outs are distinguished from the outputs, as the in/outs need further processing. For example if we have four functions with the following parameters $A^{(out)}$, $A^{(in/out)}B^{(out)}$, $B^{(in)}C^{(out)}$, $C^{(in)}$. Then the VT will be: $A = \{A, A^{io} B\}$; $B = \{A^{io} B\}$; $C = \{C\}$.

2. Expand input/outputs with the outputs. Continuing the previous example, the A^{io} need to be expanded. $A = \{A, A^{io} B\}$ so everywhere A^{io} appears, this needs to be replaced with A and A^{io} B. This is done once, with no further looping to replace variables still marked as input/output. Duplicates are removed. The example VT has now become: $A = \{A, AB\}$; $B = \{AB\}$, $C = \{C\}$

3. For each function, list its variables as a set. Firstly list variables that are inputs (including input/output). Then list all variables for each function as a set. Continuing the previous example, this produces the following lists: (inputs) A, B, C and (all) A, AB, BC, C.

4. Expand the sets of variables with the variable tree. For example, every time variable A appears, it is replaced with A and AB. Remove duplicates.

Using the same example as before:

Step 1, build the VT:

$W=\{WY\}$, $X=\{XY^{io}\}$, $Y=\{WY, XY^{io}\}$, $Z=\{Z\}$

Step 2, Expand input/output parameters in the VT. Ensure self-expansions are carried out prior to other expansions using that variable. For example, Y is expanded before X:

```
W={WY}, X={WXY}, Y={WXY, WY}, Z={Z}
```

Step 3, create a list of variable sets from the functions

```
        (in and in/out)          (all)
One           Y                  XY
Two           X                  XZ
Three                            YW
Four          W                   W
Five         YZ                  YZ
```

Step 4, expand each with the values from the VT

```
Y  = WXY, WY
XY = WXY
X  = WXY
XZ = WXYZ
YW = WXY WY
W  = WY
YZ = WXYZ WYZ
```

Then remove duplicates:

```
WXY, WY, WXYZ, WYZ
```

Producing the following matrix:

	0	WY	WXY	WYZ	WXYZ
0		Three			
WY	Unset	Four	One		
WXY		Unset	Four		Two
WYZ				Four Five	One
WXYZ			Unset	Unset	Four Five

2.4 The Execution Phase

Once the tests have been generated, they are run twice: first on the oracle system and then on the SUT. Before the tests are run on each system, it may be necessary to edit the configuration file. Custom parameters can be defined whose values may need to be different depending upon whether the test is run on the oracle system or the SUT. This may be because the oracle does not implement some features in quite the same way as the SUT.

The results from running each test are written to a separate file. The output resulting from executing each TCD file is written to the file with markers delimiting the output from each TCD file. The markers include the name of the TCD file, which will be used for the comparison phase. The test results from the oracle system and the SUT are written to different locations. These test outputs become the inputs to the comparison stage.

A clean-up routine is executed between each test. This clean-up routine ensures that a failure in an earlier test does not cascade and result in many, if not all, subsequent tests failing. This cascade of failures will result in wasted test effort.

2.5 The Evaluation Phase

The evaluation phase needs to handle the differences in output between the oracle and the SUT. Some of the differences will be where there are bugs in the SUT. However, Alltest concentrates on using an oracle that is already available. Therefore the oracle is unlikely to be perfect. This imperfection of the oracle will cause expected differences that need to be handled gracefully, while the unexpected differences (which are likely to be bugs in the SUT) are flagged so they can be rectified.

Various approaches to solve the problem of selectively ignoring or matching the outputs from the two systems were investigated. One approach was to use regular expression matching to indicate the parts of the test output that should match. This

approach was discounted for two reasons. Firstly, regular expression matching is very difficult to get right. Even experienced software engineers struggle with regular expressions. Secondly, the exact output from the two systems was being specified as regular expressions for each test written. This was moving away from using the oracle system to check the results, and relying more on the regular expressions. Details of this approach are in Appendix C.

A second approach was to use "Rules". A rule is a function that attempts to make a specific change to the test output. A rule is successfully applied when it has changed the test output. Alltest applies the specified rules until the output from the SUT and oracle match or until all specified rules have been exhausted.

Initially tests were written by hand, and each rule was applied over the whole test output. This was too coarse, which meant that important details were ignored making it difficult to specify rules accurately enough. The decision was made to describe the functionality of the SUT as small units (these were the "units of functionality" described in section 2.1.1 and implemented as TCD files). The test cases were then generated from these TCD files with each rule being applied only over the output resulting from executing the TCD file. Figure 19 shows rules being used for the evaluation.

The TCD file specifies which rules can be applied to the test output. This is important as it prevents the use of rules which may allow a test to pass, when it should fail. The rules to apply are specified with the "USERULE" keyword within the second section of the TCD file.

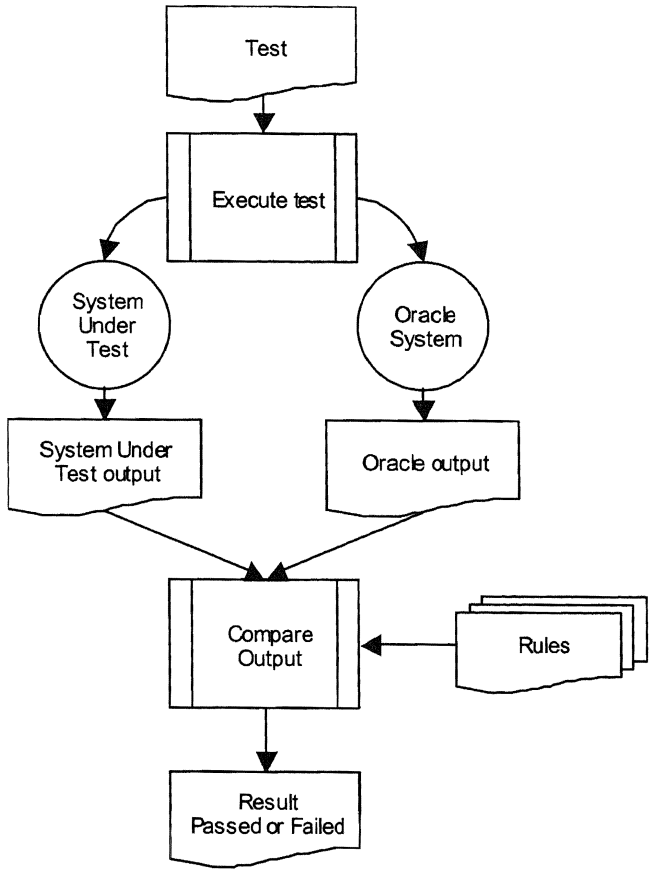


Figure 19 Using rules for the evaluation phase

Rules are implemented as functions in a scripting language. They can be defined in special rule script files or within the TCD file itself. In total there are three levels of rules that can be defined. The top level rules are "global" rules. These are rules that are supplied with Alltest as a fundamental set of rules. They may also be developed by companies using Alltest if they were testing multiple products but found some additional generic rules were useful. The second level rules are "shared local" rules. These rules are applicable to any TCD file in the interface, and would be written to allow testing of a particular product. Both "global" and "local shared" rules are defined within rule script files. Finally, there are "local" rules. These apply to a particular TCD file, and are defined within the TCD file itself.

Rules have two types: line rules which are applied to a single line of output; and file rules which are applied to the whole output for the specific TCD file. The name of

the function implementing the rule indicates both that it is a rule, and which type it is. For example `RULE_LINE__XYZ` or `RULE_FILE__XYZ`, where `XYZ` is the unique name for the rule. Other procedures in the files are ignored (they may be required to implement the rules, but are not actually the rules themselves).

A successful application of a rule will occur when a change is made to the line or file. However, it is up to the rule to correctly indicate this. Therefore, when defining a rule, the rule must specify whether it was applied successfully (the file or line was changed) or otherwise.

The comparator applies the rules in the order they appear in the TCD file. However, all the possible line rules are applied before any file scope rules are attempted.

2.6 IMPLEMENTATION

Alltest has been written using a range of different technologies. Section 2.6.1 discusses the choice of language used to implement tests and section 2.6.2 describes how tests are produced.

2.6.1 Choosing a Language for Test Cases

When developing a test suite the programming language used to implement a test is an important consideration. It is inappropriate to define a new language when well-supported and documented languages already exist. Both scripting languages and compiled languages may be appropriate. In Chapter 1 (section 1.5.3) three questions were raised regarding language choice. These questions are answered here.

Is it better to write a test as a small executable program, compiled from code, or should a test case be written in an interpreted language?

It is best to use an interpreted language to write tests as scripts. However, there are occasions when it may be necessary to use compiled code in a test.

An interpreted language avoids the overhead associated with compiling code prior to running it. Removing the need for compilation makes turn-around of test scripts much quicker, because a test can be written and run immediately without waiting for compilation. In addition, there is no need for tests to be compiled to hide the implementation of sensitive code (essential for commercially valuable applications). Speed is also not an issue. An interpreted language is fast enough for executing a test, but may not be fast enough for a released application.

Interpreted languages integrate very well with other components. They can act as "glue" to bring functionality (both interpreted script and compiled binaries) together. This makes them particularly useful for testing, as tests often need to interface to many different parts of a system (for example, the command line, APIs or specially written test interfaces).

What are the advantages and disadvantages of using an interpreted language or a compiled language for implementing the tests?

The advantages of using an interpreted language are:

- implementation speed: it is much quicker to write and immediately run a test if it does not need to be repeatedly compiled between every change.
- low complexity: as a result of not needing to include a compiler in the process when tests are being generated automatically.

- ease of use: it is often easier to write code in an interpreted language than in a compiled language.

- cross platform support and execution. Theoretically a script should run unmodified on multiple operating systems as long as an interpreter is available for the target operating system.

The disadvantages of using an interpreted language are:

- execution speed: this is often slow. However, for testing it is often not important that the tests run very quickly.

- visibility of code. A script written in an interpreted language is readable in any text editor. This may allow other people to see how the script works. For testing this is not a problem as the script is unlikely to be seen by anyone outside a particular company. In fact this could be considered an advantage, as it is much quicker to see the exact code that is running the test. There can sometimes be doubt that an executable program used for testing is actually carrying out the same functionality as the code which apparently implements it. This is because the code and the executable can become separated. Multiple copies of the executable may be available but these may not have proper versioning to link them to a specific version of the code.

The advantages of using a compiled language are:

- execution speed for areas where a small bit of functionality is used very regularly. If during testing a particular function is used a lot it is probably worth speeding up the execution time by implementing the functionality in a compiled language.

- interfacing into the low level functionality of the operating system.

The disadvantages of using a compiled language are:

- cross platform support. The test will need recompiling before it can be run on different platforms

- relationship between code and executable. This can often be broken. It is important to properly version compiled tests so that the code that implements the test can be guaranteed to have resulted in the executable. This adds overhead and can be difficult to achieve reliably.

Are there benefits to combining interpreted and compiled languages when writing tests?

It is sometimes beneficial to be able to use compiled code to carry out some activities. For example, some operations may be particularly time consuming if they are written in an interpreted language. If testing carries out these operations regularly it may be useful to write these operations in a language that can be compiled to produce an executable or library. Alternatively some operations may only be possible from compiled code (for example, access to low level operation system commands, or to API functions to the SUT).

Therefore, there are occasions when combining interpreted and compiled code is beneficial and sometimes essential. The choice of interpreted language may affect how easy this is to do. For example, Tcl can be extended by writing functions in C that are made available to the interpreter through libraries. Within a script it is impossible to tell whether a library function is being called from an interpreted or compiled library.

Defining Test Cases in Alltest

The decision was taken to use an interpreted language to define the tests in Alltest. Tcl was chosen for a number of reasons. Firstly, Tcl is a widely available scripting language that is supported on Windows, Unix operating systems such as Linux, and

Macs. Secondly, Tcl is extremely flexible. It can be easily extended to produce a specialist interpreter or it can be directly embedded into other applications. Libraries can be written in C or Tcl allowing a developer to write functions for Tcl applications in the most appropriate language technology.

Alternative scripting languages include Perl and Python. These are both extremely popular, especially within the Linux developer community. Table 6 provides a summary of the strengths and weaknesses of the languages in relation to the proposed use as a language to support a test suite. The features discussed are:

- Language goal – the original focus of the language
- How easy the language is to learn. The language needs to be easy to learn to avoid discouraging people from using the test suite.
- How easy the language is to maintain. Maintenance of test scripts is a significant problem. A language that allows programs to be easily understood sometime after they were written will make maintenance easier.
- How easy the language is to extend. The language chosen needs to be extended with special functions written in C/C++ that enable interface functions from the software being tested to be called from within scripts.
- How easy the language is to embed. The suite developed needs to call the scripting language for the tests. The best way to do this is to embed the interpreter into the test suite.
- Platform support. It is important to select a scripting language that has good cross-platform support. Using a language that works best on a single platform will prevent the methodology being used on other platforms.

Perl is too Unix centric to be a good choice as a scripting language for a cross-platform test suite. Python is potentially a good choice, especially as the syntax has a lightweight feeling. However, Tcl was developed to be extendable and embeddable (ideal for use in a test suite). In addition it has a wide user community on Unix and Windows, which shows there is good cross-platform support.

Alltest has been written using a combination of C, Tcl and Lex/Yacc. The comparator and the rules written for the evaluation phase are in Tcl. Tcl is also used as the language to write the executable section of the TCD files. However, Tcl is not sufficient on its own for implementing Alltest. The TCD files need to specify which rules are used during the evaluation phase and what are the inputs and outputs to the TCD file. LEX and YACC have been used to implement a language parser to process this information in the TCD files. Finally, C has also been used to implement Alltest. In particular the algorithms used to create the Markov Chain Transition Matrix and then walk the matrix are implemented using C.

Language	Language goal	How easy is it to learn?	How easy is it to maintain?	How easy is it to extend (using C/C++)?	How easy is it to embed (within C/C++ programs)?	Platform support
Tcl	Designed as an embeddable and extensible generic language.	Very easy if C is already known.	Very easy with a readable style.	Designed for this purpose.	Designed for this purpose.	Originally UNIX but good user community on Windows.
Perl	Designed as a text processing language.	Easy if Sed and Awk are known.	Difficult as it has a particularly unreadable style.	Adapted to allow this.	Adapted to allow this.	Good UNIX support. Available on Windows, but not always reliable.
Python	General purpose object oriented scripting language.	Online documentation and tutorials available. Appears to be a straightforward language to learn.	Very readable. Indentation delimits statement groups. Many see this use of indentation as a serious weakness.	Adapted to allow this.	Adapted to allow this.	Good on UNIX. Interpreter also available for Windows.

Table 6 Comparison of Tcl, Perl and Python

2.6.2 Producing the Test Cases

The generator produces two types of output: the test cases and a library file (see Figure 16 page 77). These files are written in Tcl. The generated library file contains a procedure for each TCD file, with the name of the procedure corresponding to the name of the TCD file. Alltest provides a mechanism for executing the tests without including the actual script from the executable part of the TCD file in either the library or the generated test file. This is an abstraction that means that Alltest does not need to understand the executable section of the TCD files, as long as an interpreter is specified. This means that the executable part of the TCD files could be written in any scripting language (such as Perl), though Tcl is currently the only language supported. However, Alltest has been optimised to use an embedded Tcl interpreter so that it can directly run the executable part of the TCD files without sending the script to an external interpreter. This optimisation significantly reduces the time taken to run the tests. It is possible that other scripting languages may allow the same optimisation to be made.

The test files that are generated use the procedure names from the library file to execute the tests. Each of these procedures calls RunFunc (also in the library file), to read the corresponding TCD file and actually execute the test through the selected interpreter.

RunFunc is shown in Figure 20. It loads alltest.dll which allows a Tcl program to use the same parsing abilities of the main Alltest executable to parse the executable section of a TCD file. Alltest.dll contains the procedure At_execute, which is the Tcl entry point into the library. At_execute reads the text from the executable part of the TCD file and runs it through a Tcl interpreter.

```
proc RunFunc {filename funcname results args} {
    load "C:\\alltest\\tests\\alltest.dll"
    set cfgname "C:\\alltest\\config.txt"
    upvar $results res
    set tcl "C:\\alltest\\interface"
    set tcl [file join $tcl $filename]
    puts ">>> $filename BEGIN <<<"
    set execlist [list AT_execute $tcl $cfgname res]
    set execlist [concat $execlist $args]
    if [catch {eval $execlist} tmp] {
        puts ">>> $filename EXECUTE FAILED <<<"
        puts $tmp
        return 0
    }
    puts $tmp
    puts ">>> $filename END <<<"
    return 1
}
```

Figure 20 RunFunc procedure from the generated library

Figure 21 shows two procedures generated from the TCD files. Alltest automatically creates these procedures, so it is not expected that they should be easy to read.

The first procedure shown in Figure 21 is "NewDirName". It corresponds to the TCD file NewDirName.tcd. This function has two parameters, FileName and FileCount, both are in/out parameters. The first two lines in NewDirName handle the parameters so that they will contain new values when this function returns. (This is how Tcl handles by-reference parameters). The next line calls RunFunc, with the name of the TCD file and the name and current values of the input parameters. Finally, this function checks the result returned by RunFunc, and sets the return values of the parameters.

The second procedure shown in Figure 21 is "movefile". It corresponds to the TCD file movefile.tcd. In this case, both the parameters are inputs only. Therefore the processing in movefile is much easier. It simply calls RunFunc with the name of the TCD file and the name and current values of the input parameters. There is no other processing required.

The clean-up is also implemented in Tcl, allowing it to be run through a Tcl interpreter. It may be necessary in the future to expand Alltest to allow any

executable to be run as the clean-up. If the clean-up cannot succeed for some reason, it returns a failure that stops Alltest from executing any further tests. In practice the failure of clean-up may indicate the presence of a bug in the SUT or oracle.

```
proc NewDirName { FileName FileCount } {
    upvar $FileName ___GENFileNameNEG___
    upvar $FileCount ___GENFileCountNEG___
    if [RunFunc "C:\\alltest\\interface\\NewDirName.tcd" "NewDirName" results \
      FileName $___GENFileNameNEG___ FileCount $___GENFileCountNEG___ ] {
        if [info exists results] {
            upvar $FileName temp
            set temp $results(FileName)
            upvar $FileCount temp
            set temp $results(FileCount)
        }
    }
}

proc movefile { FileName2 FileName1 } {
    if [RunFunc "C:\\alltest\\interface\\movefile.tcd" "movefile" results \
      FileName2 $FileName2 FileName1 $FileName1 ] {
        if [info exists results] {
        }
    }
}
```

Figure 21 Procedures in the library file linking the TCD files to the test scripts

2.7 MANAGEMENT OF THE TEST PROCESS

Automated software testing results in a large amount of data to manage. There is the model that is used to produce the test cases, the tests that are created, the output from running those tests (in Alltest's case this output is from both the oracle and the SUT), the results from evaluating the output and any bugs that are found. This section discusses the different ways this data can be managed, and how this affects the creation, execution and evaluation of the test results.

2.7.1 Static and Dynamic Test Management

When managing the testing process in a static manner, each phase of generating the tests, running the tests and evaluating the results, is carried out independently. The processing is batched. This produces a static suite of tests that does not respond to

potential feedback as a result of running the tests. Figure 22 shows the workflow for a batched testing process.

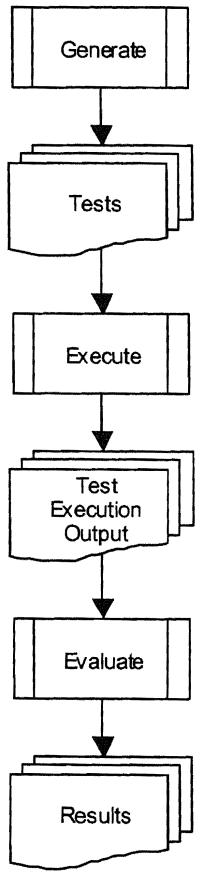


Figure 22 Managing tests with batch processing

Dynamic test management handles a single test at a time. Each test is generated, run and the test outcome is established. The results from running the test are then fed back into the generation of the next test. Figure 23 shows the workflow for dynamic test management.

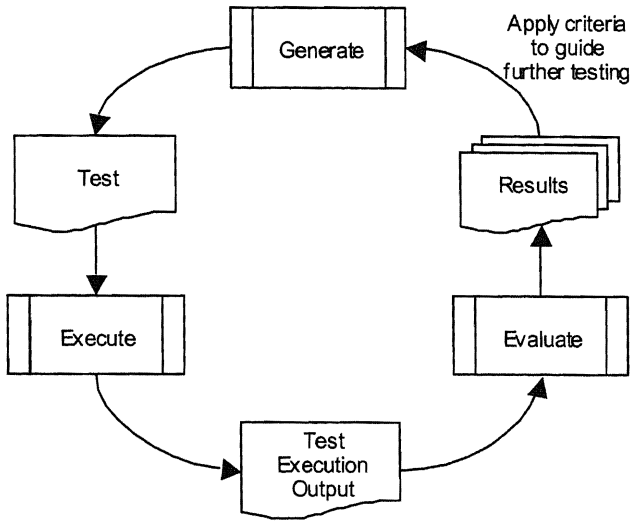


Figure 23 Managing tests using dynamic generation and execution

2.7.2 Management of the Test Data Repository

The test data repository is a database that contains test data, expected results and actual results from running the tests. It needs the following features:

- available to everyone working on a development project.
- capable of managing the storage of test scripts.
- able to allow the selection of one or more of the tests to execute.
- capable of storing multiple test results.
- able to operate over a range of platforms.

One view of the Test Data Repository is as a central database or repository (for example, in a configuration control system such as CVS). This view can be used for the following discussion on managing the Test Data Repository.

There are four ways to manage the test process, depending upon whether the tests are created statically or dynamically and if the expected results are generated at the same time as the tests are run:

Option (1): fully dynamic management. Each test is generated one at a time. The results from testing can be fed back to influence future test generation. The test data is stored in the repository for future use. The expected results are generated one at a time for each test. Figure 24 shows database access for a test suite using dynamic management.

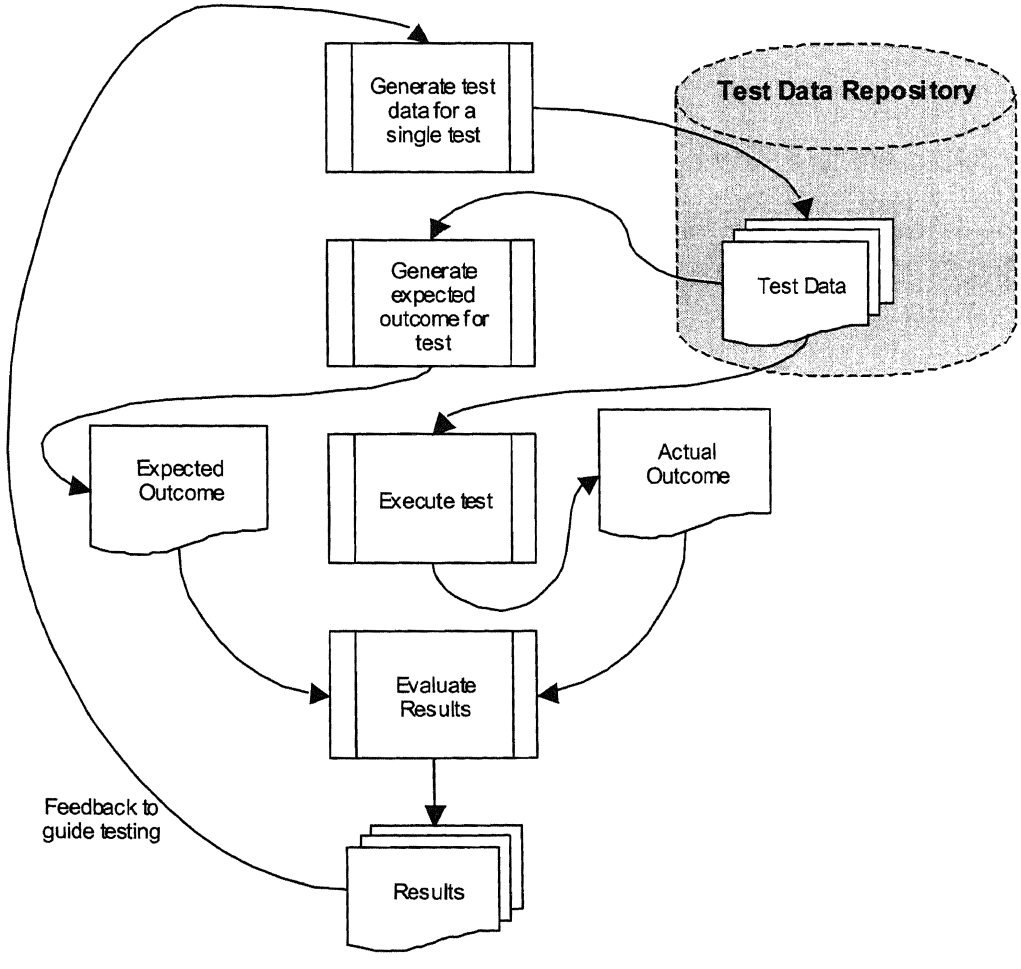


Figure 24 Test management for dynamic testing

Option (2): basic static management. All the tests are generated in one go. Extra tests can be created, but there is no feedback from the results to use when creating more tests. The expected results are generated on the same system as that used for running the tests. The expected results can be used repeatedly when re-running tests, or they can be regenerated if required. The expected results are not stored in the Test Data Repository. Figure 25 shows database access for a test suite which uses basic static management.

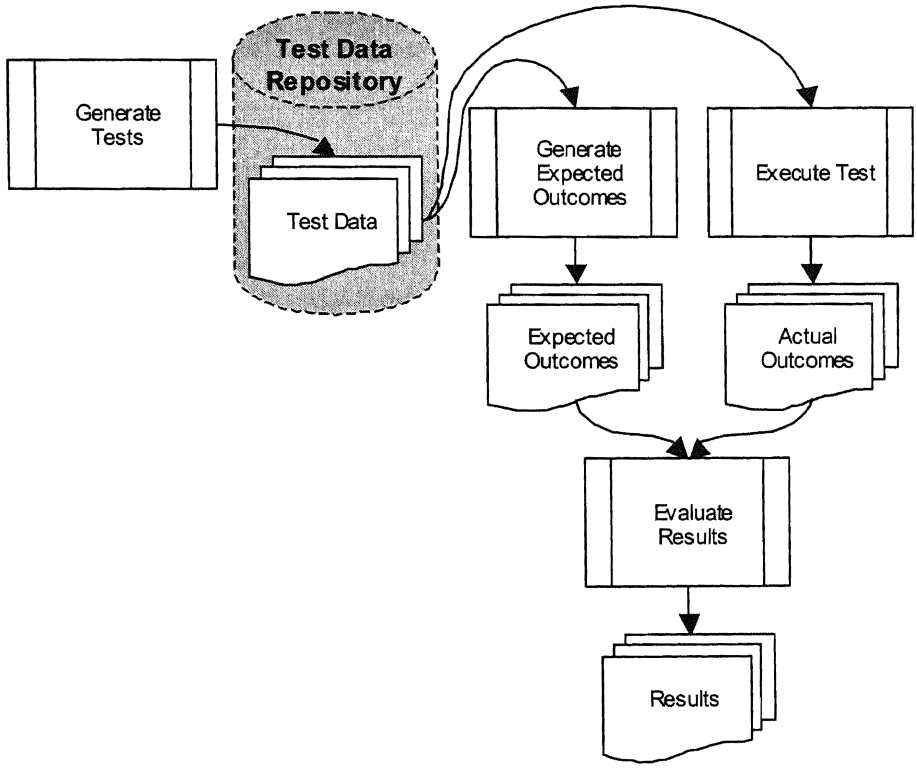


Figure 25 Test management using generated expected outcomes

Option (3): static management with stored expected results. All the tests are generated in one go and stored in the Test Data Repository. The expected results are generated and stored in the repository with the test data. The system running the tests then uses these results when evaluating the actual test results. See Figure 26.

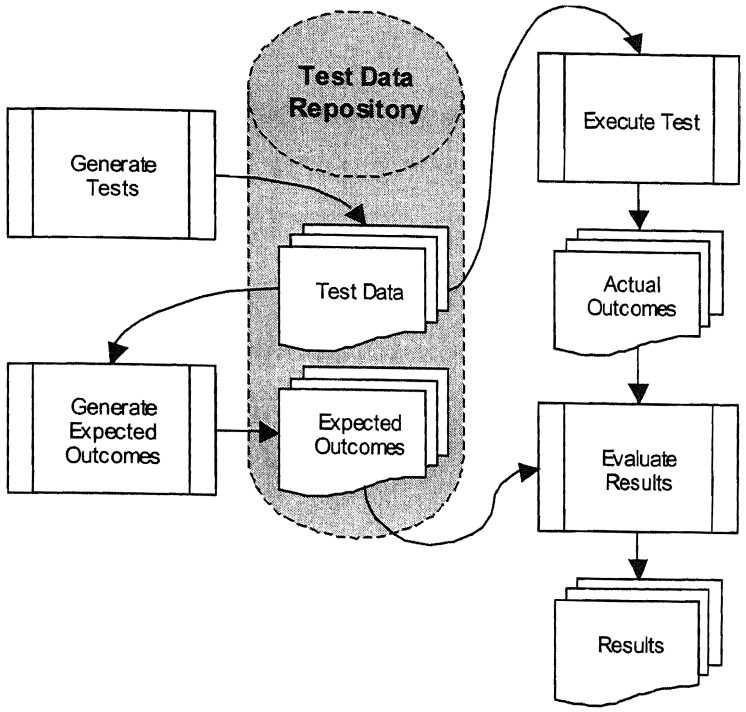


Figure 26 Test management using stored expected outcomes

Option (4): static management with two sets of expected outcomes. All the tests are generated in one go and stored in the Test Data Repository. A "Master" copy of the expected results are generated and stored in the repository (this is basically the same as option (3)). The system running the tests generates its own version of the expected results, as for option (2). The expected results generated and the stored master expected results can be compared. If there are any discrepancies these can be flagged or automatically corrected. If there are any discrepancies during evaluation of the test output, then the master versions stored in the database may also be consulted. See Figure 27. This deals with the case where the host computer system being used to run the tests may have a problem that prevents all the tests from running properly. If the results from both the Oracle system and the SUT are generated on the same host computer system, this problem may not be detected. However, keeping master versions of test results requires more effort to maintain a usable test suite. It would be much simpler to add a short test at the beginning of each test run that confirms the host computer system is correctly configured and working.

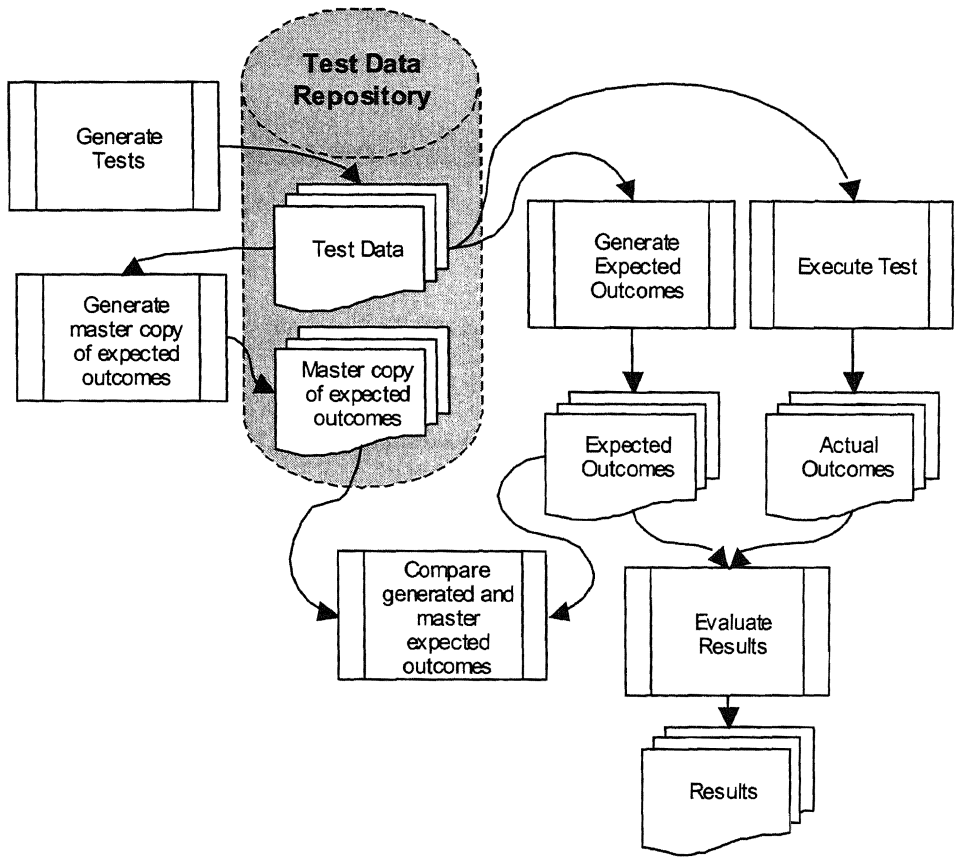


Figure 27 Test management using two sets of expected outcomes

The advantages and disadvantages of these approaches can be evaluated with respect to four factors (time, confidence, flexibility and simplicity):

Time. The time taken to generate the expected results and run a complete set of tests. Option (3) produces the best result in terms of time spent during testing, because the expected results are generated just once, at the same time as the test data.

Confidence. Option (4) allows full confidence in the results generated. Any environmental issues are handled by creating a new set of expected results for the current test configuration. The expected results can be checked against the master results either prior to running the tests or if discrepancies arise while running the tests.

Flexibility. Option (1), the dynamic method is the most flexible. It allows results from executing one test, or the set of tests so far, to feedback and influence future test generation.

Simplicity. Option (2) is the simplest. A new set of expected results is generated for the system running the tests. These are then used, and re-used while running the same set of tests on the SUT. There is no issue involved with the synchronisation or re-synchronisation of test results in the test data repository if the set of tests is added to or modified during maintenance.

Table 7 gives an indication of how the four approaches compare when assessed against each of these four parameters. The overall column adds up the number of stars for each of the four parameters to give a new star rating (0-3=●, 4-5=☆,6-7=☆☆ and 8-12=☆☆☆). As can be seen, option (4) performs extremely poorly because of the time taken and the complexity of managing testing via this approach. Options (2) and (3) perform about equally. The choice between them depends upon whether simplicity is the overriding factor (in which case option (2) is the best) or if confidence in the result comparison is the overriding factor (in which case (3) is the best approach). There is always going to be a trade off between the different factors. However, in many engineering environments simplicity is likely to be the winner. Mistakes are more likely to happen using a process that is excessively complex.

	Time	Confidence	Flexibility	Simplicity	Overall
Option 1	☆☆	☆	☆☆☆	☆	☆☆ (7)
Option 2	☆☆	☆	☆☆	☆☆☆	☆☆☆ (8)
Option 3	☆☆☆	☆☆	☆	☆☆	☆☆☆ (8)
Option 4	●	☆☆☆	☆	●	☆ (4)

Key: ☆☆☆ Excellent ☆☆ Good ☆ OK ● Poor

Table 7 Comparing the different test management options

A static approach to managing the testing process has been chosen for Alltest. Tests are generated separately from the task of running the tests and evaluating the results. This approach was chosen because batched processing of tests was a simple option, especially as there was no requirement to feed results from the testing back into the start of the process for a subsequent generation of tests. However, in the future Alltest could be changed to allow dynamic test management. This would facilitate research into test generation based upon previous test runs.

The test data repository of Alltest can be managed in any of the three static forms discussed in this section. However, during this research the test results have been generated as required, so option (2) has been employed.

2.7.3 The Bug Tracking Database

Bugs that have been found need to be managed. This is usually done with a bug tracking database. The bug tracking database is used to record the bug when it is first found. As the bug is investigated and eventually fixed, the database is used to track the bug's status. Bug tracking databases can be purchased off the shelf or can be developed using a standard database application. The latter option allows the database to be customised fully to the needs of the organisation, taking account of workflow and procedures. It also allows the database to be integrated with other applications used in development, such as the test suite and code versioning systems such as CVS.

The author developed a bug tracking database for the team developing and maintaining Manager. This was written using Lotus Notes and provided a solution for over five years[1]. It was designed as a stand alone application and does not integrate with any other development tools. Despite this, everyone developing,

[1] The development team have now moved to using Bugzilla, an open source bug tracking system. One of the primary reasons for this move was the continued resistance to using a system based upon Lotus Notes. Another reason was the perceived time needed to maintain and develop an in-house solution to managing the bugs.

testing, maintaining and supporting Manager has used the Lotus Notes bug tracking database for over five years.

A bug tracking database can be used to provide information on the types of bugs discovered and whether there are common sources of these bugs (for example, the same errors being repeatedly made in the code, or parts of the SUT, resulting in a disproportionate number of bugs). To this end a bug taxonomy was added to the database. The taxonomy was based upon IEEE Standard Classification for Software Anomalies (IEEE 1994). This gives a classification scheme for bugs, specifying mandatory and optional fields. For example, one part of the classification scheme is "Project Activity", which is broken down into nine categories. Another part is "Project Phase" which is broken down into six categories, most with at least four subcategories.

This IEEE standard is designed to be a general taxonomy applicable to any software development organisation. However, developers using this Lotus Notes bug tracking database struggled to use the taxonomy, frequently requiring guidance as to the proper classification of bugs. A second problem was that the taxonomy required the bug to be classified on a number of levels, resulting in a large number of fields. This acted as a disincentive to complete the classification accurately. As a result, though the taxonomy remained until Bugzilla replaced the database in Spring 2003, bugs entered into the database were frequently left unclassified.

The author has a number of conclusions regarding the introduction of a taxonomy into a bug tracking database. The first conclusion is that though a standard taxonomy should be applied, it is important to modify the taxonomy to reflect the terminology and processes used by the company. The second conclusion is that for the taxonomy to be successful some effort must go into training developers and testers to use the taxonomy properly. The enforcement of proper use of such a taxonomy may prove to be a challenging management problem. The taxonomy should also be very easy to fill in, with only a small number of fields. Finally, there should be a clear benefit to the developers and testers to fill in these fields. For

example, completion of the taxonomy could be included as part of their performance metrics.

2.8 How to use Alltest

This chapter has described Alltest, explaining how it automates all three phases of the software testing process. This section concludes the chapter by discussing how Alltest is used in practice.

Once the Interface has been written and tested (see section 2.1.5) Alltest can be used to test the SUT. Figure 28 is a flowchart showing the steps taken when using Alltest. The first action is to configure Alltest ready for generating tests. The values that need setting are: `markov_probability_end_test`, `markov_probability_same_state` and `generator_num_tests`. These values control the flavour of the tests produced and how many tests are created. The charts and equations in Chapter 3 show the effect of altering these parameters. In practice `markov_probability_end_test` should be set very low (less than 0.2) for tests of any meaningful length to be generated.

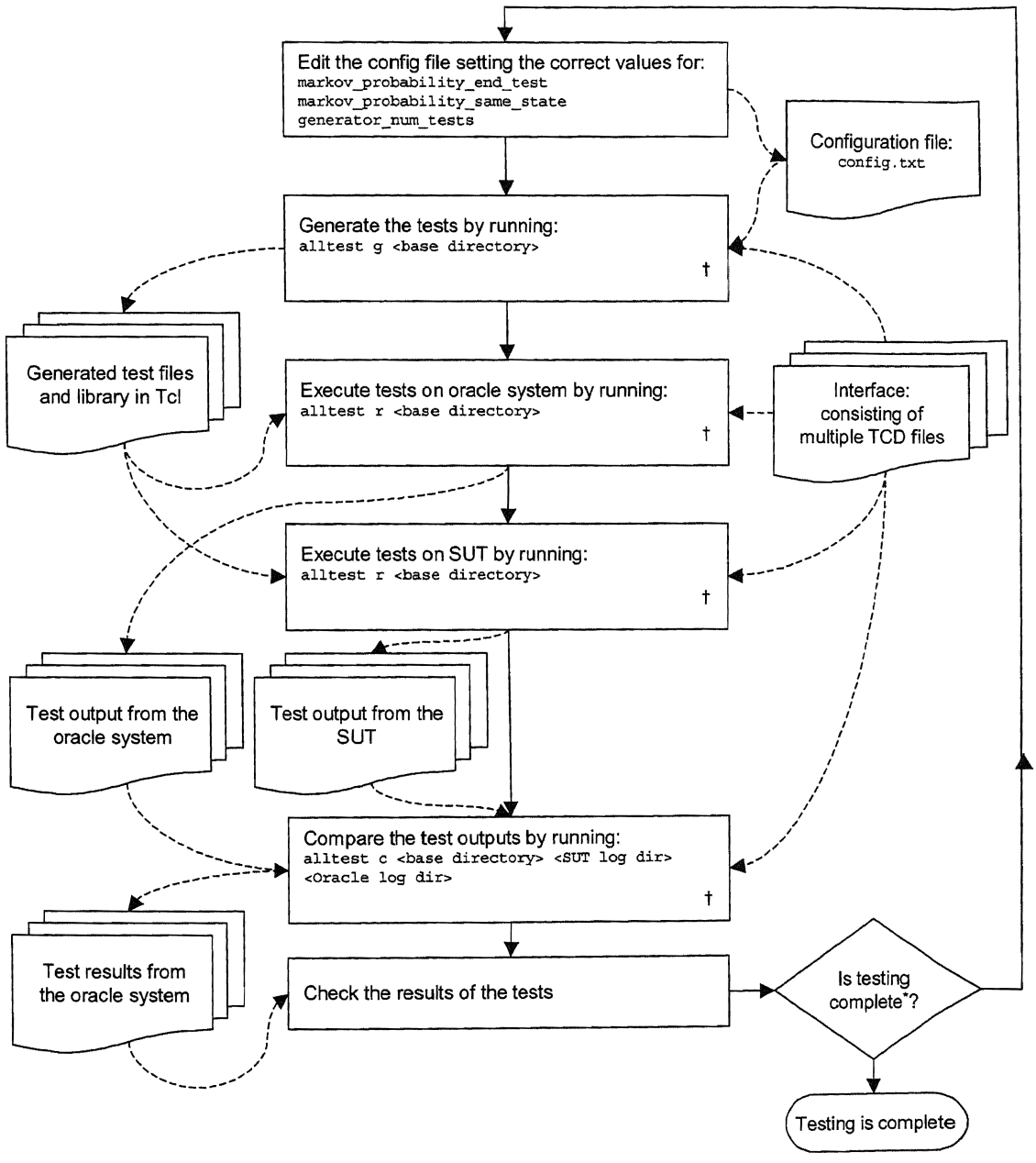
Once Alltest is configured, the tests can be generated. If the generation is run a number of times, then the tests already created are kept and the generation creates additional tests. For example, if `generator_num_tests` is set to 100, and the generation is run ten times, 1000 test files will be created. The generation is different each time, so file 1 will not be the same as file 101, etc.

Once the tests have been generated, they are executed twice: once on the oracle and once on the SUT. The configuration file may need editing prior to each test run to set custom parameters that may have different values for the SUT and oracle.

The next step is to run the evaluation. The configuration file may need editing again to ensure that parameters for the SUT and the oracle are specified correctly. The tests are run on the oracle system and the results are saved into a directory. The tests are then run on the SUT and the results saved into a different directory.

Finally, the tester needs to check the results of the comparison. A summary file is produced: `overall.log`. This file lists every test and whether the test passed or failed. It also includes a summary of the number of test results compared and the total number that passed or failed. It contains no other detail. Next the tester looks for tests that have failed and opens the appropriate test log (e.g. `test0.log`), and searches for the word "FAILED". This will find any places where the comparison failed to match the outputs from the SUT and the oracle system. The log file shows the output from the SUT and oracle tests following the application of the rules. It may be necessary to open the corresponding SUT and oracle output files and find the matching segment to see what the original output from the tests was before the application of the rules.

At this point there are two possibilities. The first is that a real problem has been found and needs investigation. The second possibility is that the rules have not been successful in ignoring output that should be considered identical. At this point testing moves into a debugging phase, either on the interface or on the SUT.



* Checks for completeness of testing include:

1. Have the tests produced adequate coverage? This may be code coverage of the SUT, code coverage of the oracle or coverage of states in the transition matrix.
2. Are more tests required that are longer, shorter, or have more or less depth
3. Were any failures found in the SUT, and as a consequence is further testing or re-testing required?

† These commands launch the automated testing processes

Figure 28 Using Alltest to generate and execute tests, and check results

CHAPTER 3 EVALUATION AND DISCUSSION

This chapter presents the experimental design, execution and results from the experiment and a discussion of those results.

3.1 EXPERIMENTAL DESIGN

In order to test the methodology three pieces of software are required. The first piece of software implements the methodology. This is Alltest and it has been described in full in Chapter 2. The second piece of software is the System Under Test (SUT). Existing commercial software, called Manager, has been selected as the SUT, and is described below. Finally an oracle system is needed. NTFS has been chosen as the oracle, and is described in section 3.1.2.

3.1.1 Manager

Manager is complex commercial systems software that controls optical data storage libraries. Manager controls the library changer, which is the robot in the library that moves disks from storage slots into drives. Once a disk is in a drive, Manager writes data to, or reads data from, the disk. Manager includes file systems, a number of device drivers, a process to control the changer in the library, a database to store the directory structure of the disks in the library, a graphical user interface (GUI) and an Application Program Interface (API). Manager is written in 'C' and consists of over 300,000 lines of code, excluding the GUI. Hence Alltest was run against a substantial piece of software. Manager's file system functionality has been targeted for this experiment. This does not use the GUI or the API (except for test set-up). However, it does exercise most of the threads and device drivers described in the next two paragraphs.

The main Manager executable consists of the following threads: Main process (this creates all other processes at start-up); one Changer process for each changer (i.e. jukebox or library) being controlled by Manager; one Drive process for each drive

being controlled; Timer process and Watchdog process. The Main process includes the process queue, which performs the queuing and scheduling of tasks that Manager is servicing.

There are also four device drivers running in the kernel: COMS, PFS, SCSI and BUFFERS. COMS provides inter-process communication between the different threads and other processes. PFS is a file system driver. This interacts closely with the operating system to provide a single drive letter to access the library. SCSI is a storage class driver that claims SCSI devices for Manager to control. BUFFERS provides a mechanism for transferring data between PFS (in the kernel) and the drive process (in user land).

Manager is complex because of the number of interacting components, and because it interacts with low-level operating system services. Its complexity and the fact that it is commercial, make it an ideal candidate as a test bed for evaluating a novel testing methodology.

Manager's file systems have undergone a great deal of manual testing and have been used for many years by customers. Manager supports three file system formats for reading and writing data: Plasmon, UDF (Universal Disk Format) and AFS (Archive File System). Of the three formats Plasmon is the oldest, and AFS is the newest. Plasmon is an early implementation of the ECMA 167 standard. This standard is the predecessor of the UDF standard. AFS is a proprietary file system that implements WORM behaviour on all disks. The file system tested was "Plasmon".

3.1.2 NTFS

In order to test the file system behaviour an oracle was needed that performed similar functionality. In this case there are many possible oracles to choose from. Manager's file systems should appear just like a native file system to any

application that uses them, so NTFS, a file system on Windows 2000, has been chosen as the oracle.

3.2 CHECKING ALLTEST

Alltest has been thoroughly tested during its development. Unit testing, module testing and system testing have all been carried out to ensure that Alltest is reliable software. However, additional heuristic checks can be carried out to confirm that Alltest generates tests as expected. These additional checks are explained in this chapter.

The generator uses two parameters to alter the type of tests it generates. These parameters are:

1. The probability of ending the test

2. The probability of staying in the same state

Some statistics gathering code has been added to the generator. This allows the capture of data to enable the examination of the effect of altering the probabilities of ending the test and of staying in the same state. The statistics code assumes that any TCD files that set an output are "Initialisation" calls, and any TCD files that only have inputs are "Action" calls. For the set of TCD files created to test Manager these assumptions are true.

For each point, 1000 tests were generated and the statistics noted from these. Appendix F shows the values used to generate the tests, and the mean and standard deviations of the numbers of transitions and types of transitions in each test generated.

An estimate for the average length of a test, which is the number of transitions (t), can be calculated where p = probability of ending the test on this transition.

$$t = \frac{1}{p}$$

The calculation is in Appendix G.

The graph in Figure 29 shows the effect of changing the probability of ending the test on the actual test length. For clarity the graph is plotted logarithmically. No other properties (such as number of change states) are shown. This is because the probability of ending a test has a very strong influence on the test length, so that when the probability is 0.15 or above, the tests are all so short as to be meaningless. The line shown on the graph is $t = p^{-1}$.

Figure 30 shows the result of varying the probability of staying in the same state (q). The equations for the lines shown on Figure 30 are:

Number of transitions (t)	$t = 1000$
Transitions that remain in the same state (s)	$s = 1000q$
Transitions that cause a change in state (c)	$c = (1\text{-}q)1000$

The average number of transactions (t) is $t = p^{-1}$ (as above). For the set of tests generated p is set to 0.001. Therefore $t = 1000$ for all these tests.

The values of t, s, and c, are related such that $t = s + c$. s is directly proportional to q, and c is inversely proportional to q giving the equations $s = 1000q$ and $c = (1\text{-}q)1000$. These equations are plotted on Figure 30, showing a good fit between these lines and the generated data points.

Trend lines are not plotted for the other three sets of data ("action" calls (a), "initialisation" calls (i) and "unset" calls (u)). This is because their trend lines are dependent on the number and types of TCD files in the interface. However, the probability of staying in the same state does influence the number of "action" and

"initialisation" calls made, and have an inverse relationship with the number of "unset" calls made. The values of t, a, i, and u are related such that $t = a + i + u$.

In Figure 29 and Figure 30, the calculated lines plotted against the recorded values show a good fit. This gives confidence that Alltest is creating and traversing the Markov Chain Transition Matrix correctly when the tests are generated.

"initialisation" calls made, and have an inverse relationship with the number of "unset" calls made. The values of t, a, i, and u are related such that $t = a + i + u$.

In Figure 29 and Figure 30, the calculated lines plotted against the recorded values show a good fit. This gives confidence that Alltest is creating and traversing the Markov Chain Transition Matrix correctly when the tests are generated.

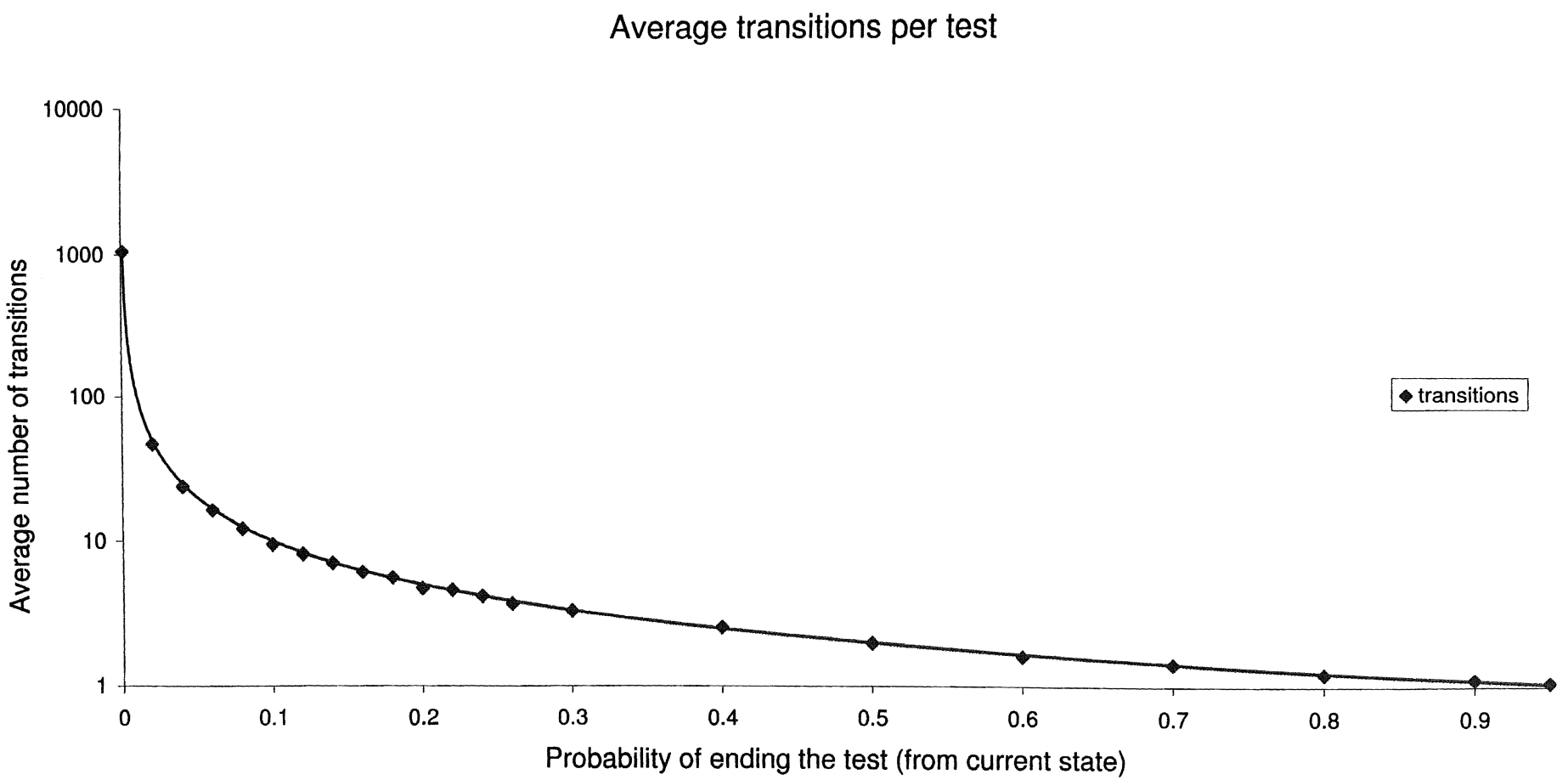


Figure 29 Effect of varying the probability of ending the test

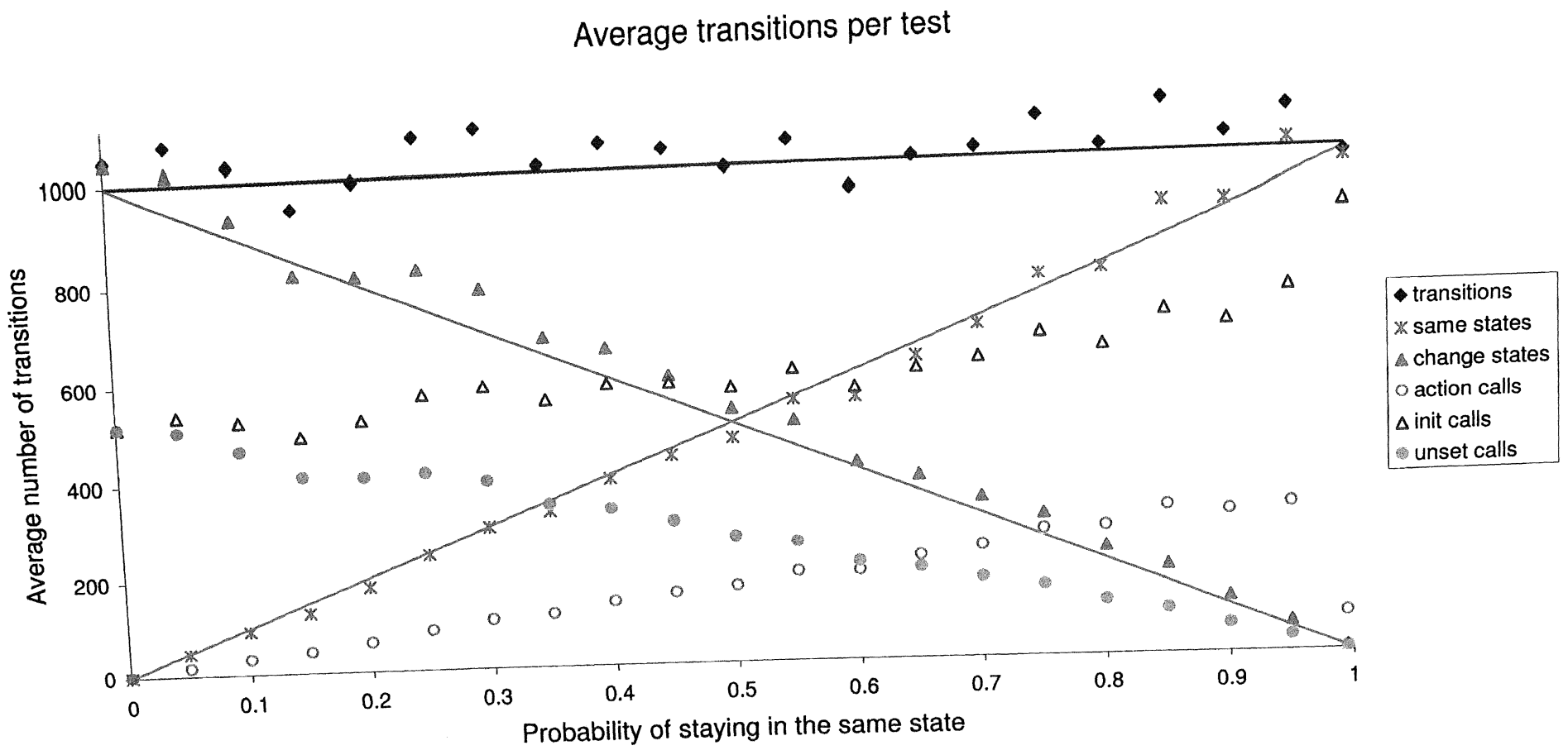


Figure 30 Effect of varying the probability of staying in the same state

3.3 DESCRIPTION OF THE EXPERIMENT

The experiment using Alltest to test Manager against NTFS has been to establish the effectiveness of the methodology and the amount of effort required to use the methodology. The methodology must be shown to be cost effective, hence the time taken to develop the interface is important. An interface has been written and various data recorded:

- The time taken to write interface files (TCD files including rules for the comparison).
- The bugs found by testing Manager with Alltest.
- The time taken to run sets of tests and to compare the results.

The person who wrote the interface was the author of this thesis. She is an experienced programmer, with more than eight years commercial experience of C, and a number of years experience of C++, Delphi, Pascal and Tcl. This should be taken into account when considering how much effort a tester or less experienced programmer would need to apply when using Alltest.

The interface contains 19 TCD files (see Table 8). Each TCD file contains a Unit of Functionality (UOF). The table shows that there are 10 files used for initialising parameters, and 9 files that actually carry out actions during tests. In the table, "Num Lines" specifies the total number of lines in the TCD file, including blank lines and comments. "Num Rules" specifies the number of rules created specifically to test Manager and does not include global rules written for the Alltest suite.

The generator produces a transition matrix automatically using these TCD files. Seven parameters are initialised in the TCD files. However, FileName is needed twice, which gives eight parameters. The resulting matrix has 20 states and is shown in Appendix A.

Initialisation Files			
Filename	**Function prototype**	**Num Lines**	**Num Rules**
BaseDir.tcd	BaseDir(out string FileName, in string TestVol, inout integer FileCount)	46	0
InitialiseCount.tcd	InitialiseCount(out integer InvFileCount, out integer FileCount)	24	0
InitialiseDelParam.tcd	InitialiseDelParam(out integer DelParams)	15	0
InitialiseSubDir.tcd	InitialiseSubDir(out integer SubDir)	15	0
NewDirName.tcd	NewDirName(inout string FileName, inout integer FileCount)	121	0
NewShortName.tcd	NewShortName(out string ShortFileName, in string FileName)	28	0
NewWildcardName.tcd	NewWildcardName(inout string FileName, inout integer FileCount)	90	0
SetDelParams.tcd	SetDelParams(inout integer DelParams)	15	0
SetSubDir.tcd	SetSubDir(inout integer SubDir)	24	0
testvolume.tcd	testvolume(out string TestVol)	24	0
Action files			
Filename	**Function prototype**	**Num Lines**	**Num Rules**
copyfile.tcd	copyfile(in string FileName2, in string FileName1)	26	0
del.tcd	del(in integer DelParams, in string FileName)	106	0
listdir_basic.tcd	Listdir_basic(in string FileName)	46	1
mkdir.tcd	Mkdir(in string FileName)	44	0
mkfile.tcd	Mkfile(in string FileName)	49	0
movefile.tcd	movefile(in string FileName2, in string FileName1)	285	1
readfile.tcd	readfile(in string FileName)	40	0
renamefile.tcd	renamefile(in string ShortFileName, in string FileName)	124	1
rmdir.tcd	rmdir(in integer SubDir, in string FileName)	46	0

Table 8 Interface files written to evaluate Alltest

3.4 Experiment and Results

This section gives a stage-by-stage account of the creation of the TCD files and rules needed to test Manager's file system. It examines the effort taken and the bugs found. Ten bugs were found despite Manager being mature and thoroughly tested software. Of these bugs, seven of them were ranked with a severity of high or very high. The timings show that only modest programmer effort was needed.

While Alltest was being implemented, nine TCD files were created (see Table 10). These files do not form part of the effort-experiment. With these nine interface files a total of four bugs were recorded:

Bug 1 error codes returned from the SUT do not match those from the oracle.

Bug 2 rd /s /q fails to remove files with very long names.

Bug 3 mkdir fails to make intermediate directories.

Bug 4 mcf_open hangs when stand-alone drive is mapped across a network.

For the effort-experiment a further ten interface files and additional rules were written and the time to develop them recorded. Table 11 shows these TCD files and rules. The following gives a stage-by-stage summary of the effort-experiment and the bugs found in Manager.

Stage 1 Create TCD files and rules to handle renaming files, moving files and directory listings. Following creation of the TCD files and rules, ten tests were generated and run and seven tests failed. An additional line was added to ListDir_Basic.tcd to USERULE LINE FA_FAILURE_MESSAGE. This resolved six of the failures. The final failure was a bug in Manager (bug 5).

With these four additional interface files the following bugs were found:

Bug 5 creating a new file failed.

Bug 6 rename file over itself has different result on SUT.

Bug 7 net use /d hangs.

Bug 8 move directory fails with "access is denied".

Stage 2 Rule added to ignore failure caused by bug 6.

Stage 3 Rule added to handle move failures that are reported over multiple lines.

Stage 4 TCD files added to handle removing a directory (`rmdir`). Initially included subdir specification within the test itself. However, it became apparent that the subdir specification should be an input parameter into rmdir. This required the creation of SetSubDir.tcd and the addition of InitialiseSubDir.tcd.

Stage 5 Add rule to handle case when move fails for both SUT and oracle.

Stage 6 TCD files added to handle deleting a file. The command line function `delete` has a large range of parameters: `/P` (prompt, not relevant for automated testing), `/F` (force deletion of read only files), `/S` (recursive deletion through sub directories), `/Q` (quiet, opposite of `/P` and always set for the automated test), `/A` (attributes: `R` read-only file, `S` system file, `H` hidden file, `A` file with archive flag set. If the attribute is prefixed with - this means "not".) In order to set these parameters the easiest option was to use bit setting and checking; and have only one parameter as an input to del.tcd.

After the six interface files were added at Stage 4 and Stage 6, the following bugs were found:

Bug 9 renaming files produces empty file.

Bug 10 rename file21 *1.* has different message on SUT.

In total ten bugs were found within Manager. Table 9 shows the number of bugs found at each severity rating. Full descriptions of these bugs are in Appendix B.

Severity	Description	Number
Very High	Causes corruption to data	1
High	Failure of feature which will impact upon customers	6
Medium	Failure of feature which can be simply worked around	1
Low	Problem with data/text output	2

Table 9 Number of bugs found listed by severity

TCD filename	File type	Description
BaseDir.tcd	Initialisation	Creates a filename, including drive letter. E.g. x:\File1.
CopyFile.tcd	Action	Given two filenames, calls the command line function to copy the first filename to the second filename.
InitialiseCount.tcd	Initialisation	Initialises the FileCount and InvFileCount parameters. These parameters are used as inputs to other TCD files.
MkDir.tcd	Action	Calls the mkdir command line function.
MkFile.tcd	Action	Creates a file of the specified name and writes a small amount of data to the file.
NewDirName.tcd	Initialisation	Given an input filename (for example, the output from BaseDir.tcd) this creates a filename. The filename may include both valid and invalid characters, may be long or short and may or may not include an extension.
NewWildCardName.tcd	Initialisation	Given an input filename (for example, the output from BaseDir.tcd) this creates a filename containing wildcard characters.
ReadFile.tcd	Action	Opens a specified file and reads some data from it.
TestVolume.tcd	Initialisation	Obtains the test volume (drive letter) from the main configuration file.

Table 10 TCD files created during the development of Alltest

Stage	TCD file or Rule name or Change	File type	Description
1	RenameFile.tcd	Action	Call the command line function to rename a file.
	MoveFile.tcd	Action	Call the command line function to move a file.
	ListDirBasic.tcd	Action	Carry out a directory listing using the "dir" command.
	NewShortName.tcd	Initialisation	RenameFile.tcd requires a filename without any path. This TCD file takes a filename with a path and removes the path except for the last element.
	RULE_LINE__IGNORE_DATES	Rule	Ignores dates, as these are only valid if files are created at exactly the same time. Uses regular expression matching which is difficult – hence the time taken to complete rule
	RULE_LINE__IGNORE_BLANKS	Rule	Ignores blanks.
	RULE_LINE__IGNORE_VOLUME	Rule	Ignores the volume.
2	RULE_FILE__RENAME_IGNORE_SELF	Rule	Added to RenameFile.tcd. This rule uses regular expression matching, which is difficult to get right. Initially this took 1 hour 32 minutes to write. However, the file needed to be adjusted to handle the full path correctly, adding 40 minutes to the development time.
3	RULE_FILE__MOVE_IGNORE_BOTH_FAIL	Rule	Some move failures are reported over two lines, but the existing failure handling rule only deals with failures over one line. This rule handles move failures over two lines.

Table 11 TCD files and rules added

Stage	TCD file or Rule name or Change	File type	Description
4	`rmdir.tcd`	Action	Calls the command line function to remove a directory.
	Add subdir extension to `rmdir.tcd`	Action	Handle removing sub directories
	`SetSubDir.tcd`	Initialisation	Rework subdir into a separate file, and change rmdir.tcd to handle this additional input
	`InisialiseSubDir.tcd`	Initialisation	
	Change `rmdir.tcd`	Action	Modify rmdir.tcd to print out the value of the subdir parameter
5	`RULE_FAIL__MOVE_IGNORE_BOTH_FAIL`	Rule	Handle move failing on both oracle and SUT but with a different message
6	`Del.tcd`	Action	Implements del command line function. One of the inputs is a bitmap to specify the parameters for the del command.
	`InitialiseDelParam.tcd`	Initialisation	Initialise the parameter bitmap
	`SetDelParams.tcd`	Initialisation	Randomly set the parameter bitmap

Table 11 TCD Files and rules created (continued)

The time taken to write the whole interface can be estimated by extrapolating the timings that have been recorded. Lines of code (LOC) is the measure that has been used as the basis for the extrapolation. LOC is not a very good measure of program complexity. However, for the program files being examined, it is a consistent measure. Other measures that could be used as a basis for an extrapolation are, the number of parameters in the TCD file or the number of rules used. Number of parameters may map onto Function Point Analysis (Symons 1991), and may be worth exploring for future estimates. However, the code in the TCD files takes the majority of the time to write, therefore it is sensible to use the code as the basis for extrapolating the time taken to write the TCD files. The code in the TCD files is written in Tcl, which has a consistent style. For example, an if/else statement is always written as shown in Figure 31. This means that an if/else statement (regardless of the tests to be executed, and excluding the statements to be executed) will always cover 3 lines. This is different from C, for example, where different coding styles result in very different line counts.

Table 12 shows each TCD interface file created, with a breakdown of its characteristics, including the time taken (estimated or recorded) to write the file. "Num lines" is the number of lines of code in the given procedure. It includes comment lines, together with those lines before the procedure that describe it. It excludes blank lines. "Initial comment lines" is the number of lines of comment at the top of the TCD file. "Total time taken" is rounded up to the next whole minute. Times given in brackets are estimated. Procedures called "main" are the executable lines in the TCD file that are not actually part of a procedure. Table 13 shows the same information for each rule written.

```
if [a_test] {
    # statements to execute
} else {
    #statements to execute
}
```

Figure 31 Code showing example "if / else" statement in Tcl

For the TCD files where timings were recorded:

- 129 lines (including comments, but excluding blank lines)
- 143 minutes (2 hours 23 minutes)

Extrapolating to all TCD files:

- 464 lines (including comments, but excluding blank lines)
- 514 minutes (8 hours 34 minutes)

For the rules where timings were recorded:

- 240 lines
- 282 minutes (4 hours 42 minutes)

Extrapolating to all rules:

- 552 lines
- 649 minutes (10 hours 49 minutes)

In total 19 hours 23 minutes were used to write the interface. Which is approximately equivalent to three working days.

File	Type initialisation or action	Num Parameters			Num Rules Used	Initial comment lines	Procedures	Num lines	Total lines	Total Time Taken
		Inputs	Outputs	In/out						
BaseDir.tcd	initialisation	1	1	1	2	1			20	(22 minutes)
							main	19		
CopyFile.tcd	action	2	0	0	4	1			4	(4 minutes)
							main	3		
InitialiseCount.tcd	initialisation	0	2	0	0	4			8	(9 minutes)
							main	4		
MkDir.tcd	action	1	0	0	3	1			22	(24 minutes)
							main	21		
MkFile.tcd	action	1	0	0	4	1			15	(17 minutes)
							main	14		
NewDirName.tcd	initialisation	0	0	2	2	1			94	(104 minutes)
							main	93		
NewWildCardName.tcd	initialisation	0	0	2	2	1			59	(65 minutes)
							main	58		
ReadFile.tcd	action	1	0	0	4	1			14	(16 minutes)
							main	13		
TestVolume.tcd	initialisation	0	1	0	2	2			6	(7 minutes)
							main	4		

Table 12 Breakdown of the characteristics of the TCD interface files

File	Type initialisation or action	Num Parameters			Num Rules Used	Initial comment lines	Procedures	Num lines	Total lines	Total Time Taken
		Inputs	Outputs	In/out						
MoveFile.tcd (version 2)	action	2	0	0	3	1			93	(103 minutes)
							MoveFile	5		
							IsValidName	12		
							CheckStatus	33		
							MoveDirectory	22		
							main	20		
ListDir_Basic.tcd	action	1	0	0	7	1			4	6 minutes
							main	3		
MoveFile.tcd	action	2	0	0	3	1			5	4 minutes
							main	4		
RenameFile.tcd	action	2	0	0	3	1			4	6 minutes
							main	3		
NewShortName.tcd	initialisation	1	1	0	2	1			5	4 minutes
							main	4		
RmDir.tcd	action	2	0	0	3	7			16	7 minutes
							main	9		
SetSubDir.tcd	initialisation	0	0	1	0	1			11	10 minutes
							main	10		
InitialiseSubDir.tcd	initialisation	0	1	0	0	2			4	1 minute
							main	2		

Table 12 Breakdown of the characteristics of the TCD interface files (continued 1)

File	Type initialisation or action	Num Parameters			Num Rules Used	Initial comment lines	Procedures	Num lines	Total lines	Total Time Taken
		Inputs	Outputs	In/out						
Del.tcd	action	2	0	0	3	20			73	105 minutes
							main	53		
InitialiseDelParam.tcd	initialisation	0	1	0	0	2			4	
							main	2		
SetDelParams.tcd	initialisation	0	0	1	0	1			3	
							main	2		

Table 12 Breakdown of the characteristics of the TCD interface files (continued 2)

Rule name	Associated Procedures	File	Initial Comment Lines	Num lines	Total Time Taken
RULE_LINE__IGNORE_VOLUME		ListDir_Basic.tcd		14	25 minutes
RULE_LINE__IGNORE_BLANKS		Rules\Lines.tcl		17	
RULE_LINE__IGNORE_DATES		Rules\Lines.tcl		35	35 minutes
RULE_FILE__MOVE_IGNORE_BOTH_FAIL		MoveFile.tcd		41	90 minutes
	helper_move_replace	MoveFile.tcd		9	
	helper_rule_ignore_single_check	MoveFile.tcd		36	
RULE_FILE__RENAME_IGNORE_SELF		RenameFile.tcd		80	132 minutes
	helper_rename_replace	RenameFile.tcd		8	
RULE_FILE__MOVE_IGNORE_BOTH_FAIL		MoveFile.tcd		4 added = 45	(51 minutes)
extended version	helper_move_replace	MoveFile.tcd		unchanged	
	helper_rule_ignore_single_check	MoveFile.tcd		39 added = 75	
RULE_LINE__NOCASE		Rules\Lines.tcl		5	(6 minutes)
RULE_LINE__SUB_PATHS		Rules\Lines.tcl	7	37	(113 minutes)
	subpaths_native	Rules\Lines.tcl		16	
	subpaths_tcl	Rules\Lines.tcl		11	
	subpaths_vol_letter	Rules\Lines.tcl		25	
RULE_LINE__IGNORE_NUMBERS		Rules\Lines.tcl		55	(65 minutes)
RULE_LINE__FS_IGNORE_FILEID		Rules\FileSys.tcl	1	18	(21 minutes)
RULE_LINE__FS_FAILURE_MESSAGE		Rules\FileSys.tcl		28	(33 minutes)
RULE_FILE__REARRANGE		Rules\rearrange.tcl		35	(41 minutes)
RULE_FILE__REMOVE_BLANK_LINES		Rules\rearrange.tcl		9	(11 minutes)
	helper__remove_blank	Rules\rearrange.tcl		22	(26 minutes)

Table 13 Breakdown of the characteristics of rules

3.5 EFFORT

Ten bugs were found with an interface that took an estimated nineteen and a half hours to write. Therefore each bug took nearly two hours effort to find. Effort spent isolating the bugs, finding further information and ensuring that the bugs can be repeated is not included in the time to find each bug. Such time would be required whatever method was used to find the bug, as it is an essential part of the test/debug/fix process. However, automation does help in establishing whether a test failure is repeatable. For example a test run can be repeated. Analysing the outcomes from the repeated test run against the previous outcomes will establish whether the failure occurred in exactly the same way each time (for example bug 3, the mkdir bug) or whether the same failure occurred in different places (for example bug 5, the file creation bug). This is the first step in isolating the bug, and takes very little additional manual effort for the tester.

However, considerable computer time is needed. For example, it took over fifteen hours (unattended) to run 300 tests against Manager. The time taken is the result of Manager's performance. The same tests on NTFS only took a few minutes. Manager's performance is much better when it is driving a jukebox instead of a stand-alone drive. This is because Manager's architecture is slightly different when controlling a jukebox. A database is used and there is much more caching, enabling much a better write performance. The PC used to run Manager was quite old and slow, which also impacted upon the time taken.

The comparison of the results from the 300 tests took nearly 2 hours of computer time. The slowness of the comparison is due to the method used to implement it. No time has been spent during the development of Alltest to optimise the comparison. It should be possible to make very significant improvements to the time taken for comparisons.

Analysis of the comparison results is done manually. However, time need only be spent on the tests that have been flagged as failed. The tester needs to establish

whether a failure is a real bug or is an acceptable difference, in which case the rules may need updating to compensate. If a real bug has been found the tester may wish to spend further time analysing the issue to manually reproduce the problem, or reduce the steps needed in the test to the minimum required to reproduce the issue. They may also choose to re-run a set of tests on the SUT to establish if the problem seen is random or reproducible. Time freed up from writing and executing tests means a tester will have more time to investigate issues and the development team will benefit by receiving better bug descriptions.

3.6 EFFECTIVENESS

At the start of this research project a bug-tracking database was developed. This was used to record all bugs found in Manager by the customer and by existing in-house testing. None of the bugs found by Alltest had been previously recorded in the bug-tracking database. This implies that the bugs had not been found through conventional means by the software development or test teams. This is a surprising result. Manager is mature software, which has undergone many hours of manual testing and is used by many customers.

Of the bugs found one of them was very surprising (bug 8, the move directory bug). Manager has undergone thorough manual testing and many years of use by various customers. Yet, Alltest found this bug that was simple to reproduce and should have been found with the previous manual testing.

Four of the bugs found during this experiment were the result of the oracle demonstrating unknown or surprising behaviour. In particular the error codes bug and the mkdir bug (bug 1 and bug 3) would not have been found without an oracle implementing correct behaviour. Without a good oracle it is not possible to establish satisfactorily whether the software has passed or failed a test. This fact is established in the literature, though there is little emphasis in the literature on selecting oracles.

Four of the bugs were discovered with the clean-up code or because of interactions between the clean-up code and the tests. Between the tests a clean-up routine is run that leaves the system in a consistent state. Unfortunately this clean-up routine also used file system commands. As a result some of the bugs found were due to interactions between the clean-up code and the tests generated. The original clean-up routine used "rd /s /q" to remove files and directories. However, this found bug 2 (rd /s /q fails to remove files with long names). So a more advanced clean-up routine was produced that used Manager's administrative API functions. This clean-up routine needed to:

1. Disconnect the network share (net use /d)
2. Connect to Manager (mcf_open)
3. Format the media (mcf_format)
4. Close the connection to Manager
5. Reconnect the network share (net use)

As a result of this more complicated approach a total of three bugs were found (bugs 4, 5 and 7). These bugs are the result of interactions between the test cases generated by Alltest, and the clean-up code manually written. They are also the result of interactions between different parts of the system. For example, bug 5 (creating file fails) was due to the interaction between the command used to format the media (an administrative API call) and creating a file (a standard file system function). The discovery of such bugs highlights the importance of automated integration testing and system testing. Component testing alone will not find all the bugs.

Bug 9 (rename produces empty file) is a very severe bug resulting in corruption of the file system. Such corruption could result in loss of a customer's data. The potential cost of such a data loss (either to the customer or to the supplier of the

system) could be huge. This makes Bug 9 a valuable bug to find and outweighs the nineteen and a half hours effort to write the interface.

Table 14 summarises the discussion of how the testing strategy affected the type of bugs found.

Bug number and description	Severity	Should have been found with manual testing	Automation helped	Repetition or stochastic nature of testing helped	Oracle helped
1 Error Codes	High				✓
2 rd /s /q fails with long filenames	High			✓	
3 mkdir fails	Medium				✓
4 mcf_open hangs	High			✓	
5 creating a new file fails	High		✓	✓	
6 rename file over itself	Low				✓
7 net use /d hangs	High			✓	
8 move directory fails	High	✓			
9 renaming produces empty file	Very High		✓	✓	
10 renaming file give different messages	Low				✓

Table 14 Summary of benefit of Alltest against bugs found

In total, ten bugs were found in Manager during this experiment. Eight were found within the file system operations and two were unrelated to the file system. One bug (bug 8) should have been found using manual techniques. For all other bugs found, at least one of automation, stochastic testing, or using an oracle was significant. Therefore it is reasonable to say that the methodology is successful in finding bugs.

3.7 Choosing an Oracle

It is clear from the results given in this chapter, that the methodology is viable. However, selecting a suitable oracle is a key part to successfully applying this

methodology. This section examines choosing an oracle in a commercial software development environment.

An oracle should provide as much functionality as possible, reducing the areas where another approach needs to be taken. For example it may be reasonable to aim at an oracle giving a high coverage (for example 80%) of the features being tested. Except for truly novel software, it is usual that applications or systems build upon the features of previous systems. They may have totally rewritten those features, or be building a more advanced version of some competitor software, but the core features of the software are likely to operate in a similar way. Differences may need to be handled, but this is where the interface and rules really help.

There are many sources of potential oracles. An obvious one is open-source software. There is a vast range of software available that is open-source: operating systems (Linux), web servers (Apache), web browsers (Netscape), bug tracking databases (Bugzilla) and office productivity tools (Open Office) to name a few. Software may be available either free or for extended trials, for example: email clients (Eudora) and graphics editing (Paint Shop Pro). Such software may not have all the features the SUT implements. However, they do provide an oracle which implements behaviour of many of the core features of the SUT.

Larger systems or more specialised software may pose a problem that is not addressed by open source software. In these cases it is likely that there is a competitor that implements similar software. Not all functions would be the same, but again, enough to make the approach worthwhile. There is a question regarding the ethics of this approach. However, most marketing functions will have examined competitor software closely (possibly bringing it in-house for a thorough examination), leaving the way open to use this software as part of the testing strategy.

A bigger question arises over patents. If there are features in the competitor software that are patented, then features in the SUT software should not be the

same. However, patents usually describe method rather than outcome, so this may not be an issue as the outcome is what is important. The largest disincentive to using this approach is when a company is producing software so close to that of a competitor that the competitor believes the copyright or patent has been infringed. If this happened and a competitor's software was used as part of the testing process, then it may become harder for the company to defend the case. Though the issue of ethics and patents may need considering when using competitor software as an oracle for testing, the issue really occurs when designing the system. If the system has been specified and designed independently of any competitor software, then as long as patents are not infringed, there should be no problem in using competitor software as an oracle for testing.

The topics of ethics, intellectual property and patents are very complex. For example, different countries have dissimilar rules for the granting of patents. Therefore, these topics are beyond the scope of this thesis and are not going to be discussed any further.

Finally, an oracle can be selected that is a direct predecessor. In this case the oracle and the SUT may be quite closely related. Companies often rewrite software completely, rewriting the old features while adding new ones. This may be the result of needing to redesign the software as a whole to enable it to be used in ways not originally envisaged. Alternatively, the software may be moving to a new platform or a new language that is now more appropriate. Such a predecessor is ideal for use as an oracle. A more closely related predecessor, which will be a direct ancestor (such as a previous version where new features have been added) is reasonable. However, this limits the testing to regression testing.

To summarise, oracles can be selected from a number of sources: open-source software, software from a competitor or direct predecessors.

3.8 Applying the Methodology in Practice

Unit testing is the first type of dynamic testing that can be carried out on newly developed software. It can be carried out as soon as some code, such as a function, has been compiled and is ready to run. From this, testing progresses through component testing, integration testing, subsystem testing, system testing, field testing and acceptance testing. The methodology described in this thesis can be applied to a number of these areas.

However, it is inappropriate to apply this methodology to unit testing. This is because unit testing should concentrate on the function independently of other code areas. The developer should be carrying out unit testing. They should be aiming to obtain a reasonable level of coverage, for example 100% branch coverage (Beizer 1990). An automated oracle is not required for this type of testing. The developer should know what the inputs and expected outputs are for the function under test. Straightforward approaches such as equivalence partitioning and boundary value analysis should be applied to maximise the likelihood of detecting errors in the code.

The experiment carried out with Manager and NTFS has shown that the methodology works for component testing. However, this methodology is also suited for applying to subsystem, integration and system testing. For these areas a suitable oracle (or oracles) will need to be found. If a suitable oracle is available, the methodology can be used for subsystem or system testing. Where the oracle models only part of the behaviour, multiple oracles may be needed. Further work is needed to assess the practicality of using multiple oracles. However, different oracles could be used for different components being tested.

3.9 Improvements to the Implementation of Alltest

Alltest is prototype software that implements the methodology of using pre-existing software as an oracle and automatically generating stochastic tests. The experiment

probability of staying in the same state increases, the number of initialisation calls made should fall.

The number of initialisation calls does not fall as the probability of staying in the same state rises because the Markov Chain Transition Matrix allows initialisation calls to be made, even when there is no change of state. For example, if you are in a state defined by having variable X set, it is still possible to repeatedly call "set variable X" and not change state. Repeated calls to "set variable X" will result in X being set to different values. As a result the probability of staying in the same state does not have the desired effect of allowing repeated operations on a single object.

The solution to this problem is to explicitly define "initialisation" and "action" TCD files. The Markov Chain Transition Matrix should then be built such that "initialisation" TCD files are only called when a change of state will actually result. If a change of state does not result then "action" TCD files are the only ones that can be called.

3.10 Research Aims Reviewed

Chapter 1, section 1.5 gives a set of research aims. There were criteria for a successful test methodology, objectives for the methodology developed and a set of research questions that needed to be answered, either in general or for the methodology developed.

The criteria for a successful test methodology (see Table 15) and the objectives for the methodology (see Table 16) have been met. The research questions have also been answered and are discussed in below.

with Manager and NTFS has shown that the methodology works. However, there are potential improvements that could be made to the implementation of Alltest.

Firstly, the comparison phase was rather slow. This is the result of the way each segment of the test output is written to a file and compared with a new instance of the TCL interpreter. This is obviously long-winded and significant improvements could be made by re-implementing the comparison phase in a more realistic fashion.

Secondly, two algorithms for calculating the states for the matrix have been presented in this thesis. The first of these has been used to develop the test cases and find bugs in Manager. However, upon implementation of an algorithm that checks the matrix, it was discovered that the generated states included two that could not be reached. This is a minor problem as all TCD files could still be called from the other states. However, it does introduce a memory inefficiency into the transition matrix. Therefore, a second algorithm is presented that produces a more memory efficient transition matrix. This algorithm has been implemented into Alltest but has not been used to test Manager, so it is not known whether the algorithm will have an impact upon the effectiveness of the tests.

Finally, improvements could be made to the way the different types of TCD file are treated within the Markov Chain Transition Matrix. Values recorded for the number of initialisation calls, action calls and unset calls are plotted in Figure 30. It can be seen that the number of initialisation calls made increases slightly as the probability of staying in the same state increases, while the number of unset calls decreases. The intention was that many operations can occur on a single object when there is a high probability of staying in the same state, and many different objects will be used when the probability of staying in the same state is low. If this were to happen we would expect that the number of initialisation calls and the number of unset calls were almost the same (there should be the same, or slightly more, initialisation calls than unset calls, because the move to the end test state can occur from any state in the Markov Change Transition Matrix). Therefore, as the

Criteria	How criteria has been met
Minimise the manual effort needed.	This is dealt with in the research questions (see section 3.10.3)
Maximise the likelihood of detecting an error.	Tests generated are stochastic and can be automatically executed. Testing aims to find bugs in software, so as tests are easy to generate, any tests that do not detect errors can be thrown away, and new tests created. The use of predecessor software as an oracle allows each executed test to be checked automatically. All these things together mean that a large number of tests can be created, executed and their results checked very quickly, which maximises the likelihood of finding an error in the SUT.
Minimise the likelihood of reporting false negatives.	Rules are used to allow differences between the oracle and the SUT. If a test fails, but the failure is not found to be significant, then new rules can be added to ignore that failure in the future.
Minimise the likelihood of reporting false positives.	Rules are applied to a small area of the test output – as defined by the units of functionality. By restricting where rules are applied, the possibility of a real mismatch being incorrectly ignored is minimised.

Table 15 Criteria for a successful test methodology

Objective	How the objective has been met
The methodology should result in test suites that are maintainable	This is dealt with in the research questions (see section 3.10.2)
The methodology must be usable by software testers and developers.	Techniques that involve manual creation of models or the development of formal specifications have been avoided. These techniques either involve a great deal of effort or specialist knowledge. Instead the interface is written using a standard procedural scripting language (with small additions) to describe the units of functionality of the SUT. Most software developers and testers will be able to define the UOFs and the rules for comparison.
A minimum of training should be necessary to be able to use the methodology in a productive way.	Appendix H provides a usage guide for using Alltest. It is envisaged that this is all that is required to productively use Alltest, or other software implementing the methodology.

Table 16 Objectives for the methodology developed

3.10.1 Oracle Problem

What form should the oracle take?

This research has demonstrated that predecessor software can be used as an oracle to test software. The oracle used does not need to be perfect, but a method must be available to handle cases where the SUT's behaviour legitimately deviates from that of the oracle.

The methodology developed uses an interface to define the functionality of the SUT and oracle. The interface specifies how the functionality of the two systems is accessed and rules are used to handle differences in output.

3.10.2 Maintainability

How are tests written using the new methodology to be maintained?

The methodology developed by this research does not require a suite of tests to be maintained. Instead an interface is used which is the only part of the test suite that needs maintenance.

How are changes in the software reflected in the work required to maintain the test suite?

Tests are generated from an interface containing descriptions of UOFs. If the SUT changes, then instead of each test being rewritten, the appropriate UOFs are modified. Tests can then be generated from scratch and the old ones thrown away.

How should "broken" tests be managed?

A broken test is one that no longer runs (for whatever reason). Tests often break when there are changes to the software under test. Using this methodology, either the interface is corrected (automatically correcting the tests) or the tests are

discarded and new tests are generated. This removes the requirement to examine possibly hundreds of tests to establish what they are doing and if they are at all useful.

3.10.3 Automation

How much manual effort is needed to implement, run and analyse the results from the tests?

Manual effort is needed when defining the interface to the SUT and Oracle system. Tests can be generated, executed and results evaluated, without manual effort. The only effort needed is to initiate the process. The methodology supports the debugging process by allowing tests to be repeated.

What is the minimum manual effort that can be expected?

It is impossible to fully automate the testing process, there are always going to be jobs that will need skilled input. For example, isolation and debugging of problems will need to be carried out manually. Though automation does help with reproducing bugs. The methodology developed during this research involves the creation of an interface that describes the functions carried out in the software and rules used during the comparison.

3.10.4 Procedures

What procedures should be followed when creating a test and managing the test throughout its lifetime?

The methodology developed moves away from a set of tests that need to be maintained for the life of the SUT. Instead an interface should be written, from which tests are generated. These tests can be discarded should errors not be found (either immediately or some time in the future when the test suite has become

"stale"). If tests become broken (perhaps because of changes in the SUT), the interface can be fixed instead of needing to make changes to multiple tests.

3.10.5 Implementation

The implementation questions are addressed in Chapter 2, section 2.6.1.

3.10.6 Management

How should tests be managed?

Chapter 2, section 2.7 gives four approaches to the management of tests. Alltest has been implemented using a static method to manage the tests. This is the simplest approach.

The best approach to managing tests is to treat the tests as disposable. By using an interface to abstract between the tests and the SUT, it is possible to throw away tests once they are no longer useful. This removes a huge burden in relation to maintenance effort and also helps to prevent testers becoming complacent. Having a large suite of tests that are executed regularly does not necessarily provide any useful information about the SUT. The only tests that should be retained are those that successfully find bugs.

There is never enough time to run tests that will completely exercise the SUT. Therefore, it is better that different tests are run each time, than to keep re-running the same tests that do not find bugs. This ensures that different functionality is executed in the SUT at each test run. Some specific tests may need to be kept and used in different tests runs, for example, if certain areas of the SUT are known to be buggy, or the test recently found a bug. However, the rest of the tests should be regenerated to increase the variety of tests that are executed.

What data are needed, and how are these to be managed?

Chapter 2 section 2.7 discusses the data that are needed to test software. The data are: test inputs, expected test outputs, actual test outputs, and the test result. If a test has failed this will lead to further data that needs to be managed. These data are the bug description, including details of the test environment and the status of the bug (for example, bug confirmed, bug fixed, fix tested, fix verified).

These data are stored in two different repositories (or databases). The first is the test data repository. This includes the test inputs and expected outputs. It may include the actual results from the test run. The second is the bug tracking database. This will record a bug when it is first suspected. Then as the bug is investigated and fixed, this database tracks the different stages and records information relating the cause of the bug and the fix applied.

What about management of the same set of tests over multiple platforms?

It is important when testing software that the methodology is applicable over different platforms. A great deal of commercial software is written that needs to run on different platforms, so the testing tools should cope with this. Alltest has been written using Tcl (this has interpreters available for Windows, UNIX and Macs) and C. If Alltest is recompiled for a different platform (the C functionality used should port over to UNIX or other platforms with minimum changes) then any interface developed (and tests generated by Alltest) can be used on any platform.

It may be necessary when writing the interface to consider the platforms explicitly. For example, the names and behaviour of operating system commands may not be the same on different platforms. The interface that is written may need to cope with this.

3.11 SUMMARY

This chapter has described an experiment into the effectiveness and efficiency of the methodology. Prototype software called Alltest has been written to implement the methodology. Alltest has been used to test commercial software, called Manager, against an appropriate oracle.

In total ten bugs were found in Manager. The interface took 19 hours 23 minutes to write. This means that, excluding the time needed to analyse failed tests, each bug took less than two hours of manual effort to find. The analysis of the test failures was helped by the ability of the test suite to rerun the tests.

This chapter finished by reviewing the aims and objectives for the methodology developed, and the specific research questions asked in Chapter 1, section 1.5.

CHAPTER 4 CONCLUSIONS

The previous chapters have described a methodology that uses predecessor software as an oracle. The methodology included techniques to automatically create the tests, execute the tests and evaluate the results from running the tests. This chapter outlines the main contributions to knowledge made by this research and outlines areas of interest for future research.

4.1 OUTCOMES OF THIS RESEARCH

One of the main aims of this research was to find a solution to the oracle problem. There were additional aims, relating to the applicability of the solution in commercial environments and to the maintainability of the tests.

An holistic approach has been taken to this research, such that all parts of the process of automated software testing have been addressed from the creation of the tests, through execution to evaluation of results.

This has resulted in a methodology in which a number of techniques have been developed, but which link together to produce a software testing methodology that is commercially viable for component, subsystem, integration and system testing.

Research in the area of software testing has often implicitly assumed the presence of a suitable oracle without explicitly defining what the oracle should be (Baresi and Young 2001, Weyuker 1982). For example, Whittaker and Thomason (1994, page 816) say:

> "We assume the presence of an oracle that is capable of comparing the output of P with the intended behavior, f, and correctly classifying success or failure."

There is no additional information regarding the type of oracle that would be suitable for this, nor how such an oracle may be obtained.

There has, however, been some research explicitly looking at the use of oracles in software testing. For example, Vouk (1990) has addressed the use of n-version software as a testing oracle. Brilliant *et al* (1990) examine the effect of coincident errors on the effectiveness of n-version programs, which is applicable when the oracle is one of the n-version programs. Dual-programming, where a high-level language is used to write an executable oracle, has been examined by Ghiassi and Woldman (1994). Other work has been carried out into *M*-mp testing and back-to-back testing. All of these methods require an executable model of the SUT to be written. This is a strategy that uses a great deal of resources, so is only viable when applied to small safety critical parts of a larger system.

This previous research has ignored the possibility of using predecessor software as the oracle. It is generally accepted that small utility programs can be used as oracles for unit testing, for example employing standard mathematics routines for checking implementations in other languages (Beizer 1990). However, using predecessor software as an oracle for testing a full system or components within that system has not been investigated before. Predecessors do not have to be direct ancestors of the SUT, but may be software that performs (in part) similar functionality to the SUT but has been independently created. For example, the word-processor "Word 2003" has many predecessors, both direct and indirect. Word 2000 and Word 6 are both direct predecessors of Word 2003. Word Perfect 5.1 and Star-office's word-processor are indirect predecessors for Word 2003. Yet all these software perform similar functions and could all be potential oracles for testing.

The methodology developed by the research presented in this thesis uses predecessor software as an oracle. As Chapter 3 has shown, the use of predecessor software as an oracle is a viable solution to the oracle problem. This is an important contribution to knowledge made by this research.

The predecessor software may not be a perfect oracle for the SUT. Its outputs may be different from the SUT and the way functionality is initiated may also be different. The methodology developed by this research uses an interface to handle these differences. Various approaches to handling the output were tried during this research. The successful solution uses rules that are applied to very focused parts of the output from each system. These rules allow the comparison to ignore specific parts of the test output, allowing a match to be made if appropriate.

Ensuring that the rules are only applied to the correct part of the test output required each test to be defined as a series of small pieces of functionality. These became known as Units of Functionality (UOF). For each UOF, rules can be specified that will be used by the comparator to find a match between the SUT and the oracle system. If all the rules are applied, and the comparator is still unable to match the output from the two systems, then the whole test is declared as failed, with the specific mismatch highlighted in the output files from the comparator.

The use of rules, and the method used to link rules to specific parts of the output, is key to the successful use of predecessor software as an oracle, and is an innovative technique that makes a direct contribution to knowledge.

The tests are created automatically from the interface files, and consist of a series of "calls" to each UOF. For automation to minimise the manual effort needed, a highly automated method of creating these tests was required.

Research has been carried out which uses Markov Chain Transition Matrices as usage models of software for statistical testing (Whittaker and Poore 1993; Walton *et al* 1995, Whittaker and Thomason 1994, Whittaker *et al* 2000, Poore *et al* 2000; Walton and Poore 2000a). These models are hand-built, with only the transition probabilities being applied automatically.

The research presented in this thesis has taken the use of Markov Chain Transition Matrices and provided two algorithms that can automatically produce Markov

Chain Transition Matrices from the UOFs. The additional data required by the algorithms to produce the matrices are the inputs and outputs to each UOF.

The algorithms produce a Markov Chain Transition Matrix (MCTM) that defines states based upon the combinations of inputs and outputs used by the UOFs. The resulting MCTM includes probabilities for staying in the same state, changing state and ending the test. It also includes "functions", which are calls to the UOFs. These functions can only be called from certain states in the MCTM. When these functions are called a state transition will occur if any variables that define the state in the MCTM are set or unset by the function call. The algorithms that allow the automatic creation of the MCTM from a small amount of data (the UOFs) is another major contribution to knowledge by this research.

As discussed in Chapter 1, section 1.3.4 there is little research published into the practicalities of executing tests and providing structured frameworks for managing the tests. Of the commercial and open source tools examined that support automated testing, there are three main types. The first type is capture/playback tools that allow manual actions on a GUI to be recorded (and saved) and played back at sometime in the future. The second type is frameworks that allow the ordered execution of manually created test scripts. The third type is tools that support the creation of stubs. None of these types allow the automated creation of tests. Therefore dealing with the issue of how tests are created, stored and executed is a key problem that needed to be resolved for this research.

This research has examined how the tests should be managed. A framework has been developed that allows each phase of the test process to be automated as far as possible. For the methodology to be useable in commercial development environments it is important to show how it will be managed, and how each individual technique works with the other techniques.

Finally, the holistic approach taken to this research means that each aspect of the testing cycle from test creation, through execution to evaluation of test results has

been examined. This has resulted in a methodology that uses a range of inter-related techniques to carry out each essential phase of the test process. It has also allowed the development of a methodology where minimal maintenance effort is required. Should part of the SUT change, then instead of each test that exercises that part of the SUT having to be changed, the only change required is to the definition of the UOF in the interface. If a tester feels it is appropriate they can rerun the test generation and throw away all previously generated tests. This significantly reduces the maintenance required for the test suite, solving a significant problem with test automation.

A technique that addresses the maintenance issues of automation, and a coherent and inter-related methodology for automated software testing, provide the final contributions to knowledge for this research.

4.2 Further Work

This section covers future potential research areas. Some of the future research directly evolves from the research in this thesis. However, some of the potential research areas are the result of new trends and techniques that are being developed in software engineering that may be applied to software testing, and these techniques should be investigated.

This section starts with a discussion of ways that the methodology can be developed further. This section then examines the broader areas of research within software testing and other technologies that can either be applied to software testing, or where there is room for research into software testing to take place.

4.2.1 Automatic Production of TCD Files

There is scope for further automation when writing the functional part of the interface. For example, if IDL[1] files are available for a CORBA[2] interface, these could be used to automatically produce the TCD files required for the interface. The manual steps would then be the production of the rules, and informing the comparison phase which rules to apply.

4.2.2 Automatic Initialisation of Variables

A second improvement is to remove the need for inputs to be initialised. It should be possible for the generation phase to detect the inputs that have not been initialised, and assign values to them based upon the type given in the TCD files using the variable. It may be necessary to define suitable values that such inputs can take, as part of the TCD files in which they are used. Some values will still need to be manually defined, for example, a number can easily be set automatically, but a filename (including volume and directories) may be difficult to define properly, especially if specific valid and invalid characters in the filename need to be tested.

4.2.3 Dynamic Testing

Alltest could be expanded to test in a dynamic fashion, generating each test, then executing it on the oracle and the SUT and immediately comparing the outputs. In this mode, testing should continue until the tester decides to stop the testing. This prevents available computing time being wasted, for example by allowing tests to run overnight, only being stopped when the tester returns in the morning.

---

[1] IDL means Interface Definition Language. IDL is part of CORBA (Common Object Request Broker Architecture). CORBA is used to enable separate system components to talk to each other, locally or over a network.

[2] CORBA means Common Object Request Broker Architecture. CORBA is used to enable separate system components to talk to each other, locally or over a network.

Optionally the tester may wish to have the testing stop early if the SUT is failing too many of the tests.

4.2.4 Adaptive Testing

Research should also be carried out into test generation based upon previous test runs. Often bugs cluster in certain areas of code. This may be due to the complexity of some areas, or misunderstandings arising from the original specification of the application. It would be beneficial to allow automated testing to identify such areas. The testing could then continue either avoiding the area until bugs are isolated and fixed, or concentrating tests on that area to find as many of the bugs as possible.

4.2.5 From Text Based to Graphics Based Testing

The software tested with this methodology was accessed via the command-line or API functions. However, this methodology is not limited to textual inputs and outputs. It is possible that the SUT and oracle are graphical programs. It should be possible to use this methodology using commercial "record and replay" tools that treat elements in a graphical application as "widgets". The interface can be written so that the widgets are accessed to cause actions (for example, selecting "open file" in a word processor from the menu), and results obtained (for example, text in a document), so that the generation, execution and comparison can still be carried out. It may also be possible to use "macros" or other programming features, built into the oracle or SUT to drive the functionality being tested.

4.2.6 Templates

The use of templates to guide testers in the development of TCD files and rules may be beneficial to testers. Some work should be carried out to establish the most common forms that the TCD files and the rules take.

4.2.7 Using Multiple Oracles

This research has concentrated on the idea of using predecessor software as an oracle. The use of more than one oracle has been hinted at but has not been developed. The SUT may have different areas of functionality that are best modelled by different oracles. Therefore, to test the SUT it may be necessary to have two, three or more oracles to provide the expected outcomes. This poses a problem, as the interface will need to work with these different oracles. It may be necessary to provide some linkage between the different oracles to enable the outputs from one oracle to relate to previous actions taken with another oracle.

4.2.8 Helpful Warnings

It is possible that TCD files are not included in the generated Markov Chain Transition Matrix. This occurs if there is a mismatch between TCD files that use parameters, and those that set them. The generation phase should identify such TCD files and issue a warning.

4.2.9 Evaluation of the Methodology on Different Types of Software

This methodology has been evaluated using Manager as a test subject. However, there are many different types of software where using predecessor software as an oracle may be appropriate. Research is needed to determine the types of applications that can use this methodology in a productive manner, and how much coverage predecessor software provides when used as an oracle.

4.2.10 Usage Models and Editing The Markov Chain Transition Matrix

If a usage model existed for the oracle system, then this data could be incorporated into the Markov Chain Transition Matrix to produce random tests that model expected usage of the SUT.

In order to achieve this it will be necessary to provide a mechanism to save a generated transition matrix, edit the matrix manually, and reload the matrix into Alltest. Currently Alltest creates the transition matrix automatically from the TCD files in the interface. This matrix is lost after the test generation is complete.

4.2.11 Reverse Engineered Specifications

The RES technique introduced in Chapter 2 (section 2.2.1) should be revisited. There have been (and continue to be) advancements in the computer understanding and handling of natural languages, for example the SIFT system (Lutsky 2000), that would contribute to making the RES technique usable.

4.2.12 Agents, Distributed Software and the Grid

Currently there is a great deal of research into Agent Oriented software (Zambonelli and Omicini 2004), distributed software and projects using and developing "the Grid" (Berman *et al* 2003). These techniques pose interesting future directions for testing research:

- The application of traditional and novel software testing techniques to software that has been developed using these architectures. This will be to determine which techniques are most effective for testing agent software, distributed software and software implementing the Grid and Grid applications.

- The application of novel software testing techniques that use software agents, Grid architecture or other distributed software models to test software of any architecture.

As an example of Grid or distributed techniques being applied to testing software, components or complete systems could be tested using the idle time of a computer (implemented as a screensaver or a background process). There are a number of large projects that use the huge computational power available in idle computers to

aid them. For example, the project to find a model for the climate in the 21st century (Climate Prediction 2005), or the search for extraterrestrial intelligence (SETI 2005). These techniques could be applied to the testing of software. With appropriate centralised support, distribution of test cases and subsequent collation of results, it should be possible to use the idle computing time available in any organisation to test that organisation's software products. Difficulties that need to be resolved when applying this technique include: the management of the test cases; management of test data for test runs on different software releases; and coordination of test runs. Coordination, to avoid duplication of testing, can be an issue if many computers are testing an application on a single computer or if each computer is running tests on their own copy of the software.

Agent oriented software is a very different software development paradigm to standard structured or object oriented software development. The following is a succinct definition (Zambonelli and Omicini 2004):

> "Agent-based computing promotes designing and developing applications in terms of autonomous software entities (agents), situated in an environment, and that can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages."

In short an application is written as a set of independent agents. Each agent is programmed with goals. The agents work to achieve these goals by interacting with each other. Agents may require specific techniques to test them thoroughly, or may be applied to the testing of other application software.

4.3 Review of Contributions to Knowledge

Identified in section 4.1 are three major contributions to knowledge: the use of predecessor software as an oracle; the use of rules and the method used to link rules to the specific parts of the output; and the algorithms that allow the automatic

creation of Markov Chain Transition Matrices from a small amount of data. There are also important advances in the methodology for framework software to run the tests. A number of possibilities for developing this work are identified in section 4.2.

APPENDIX A TRANSITION MATRIX

The generator will produce a transition matrix using the TCD interface files. The matrix shown here was produced with the TCD files shown in Table 8 page 118. The matrix was generated with the following parameters:

Probability end test = 0.005

Probability stay in same state = 0.8

The matrix is shown over pages 157 to 164. Figure 32 shows how the matrix fits together.

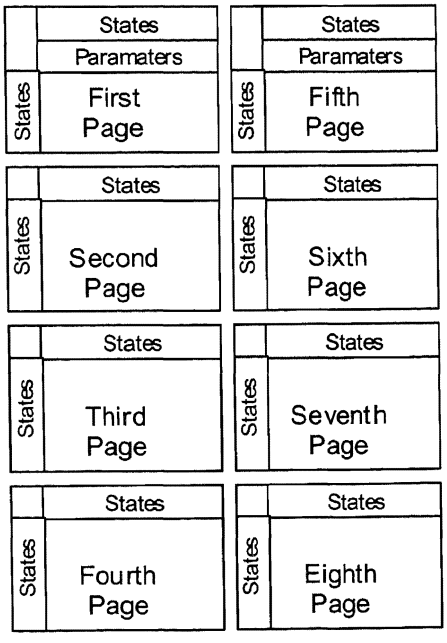


Figure 32 Transition matrix split over 8 pages

	0	1	2	3	4	5	6	7	8	9
Params		FileName1 TestVol1 ShortFileName1 FileCount1	TestVol1	FileName1 ShortFileName1 FileName2	FileName1 TestVol1 FileCount1 DelParams1	FileName1 TestVol1 InvFileCount1 FileCount1	FileName1 ShortFileName1	FileName1 TestVol1 FileCount1	TestVol1 FileName2 FileCount1	FileName1 TestVol1 FileCount1 SubDir1
0			0.249 testvolume							
1		0.8 testvolume renamefile readfile NewWildcardName NewShortName NewDirName mkfile mkdir listdir basic BaseDir						0.097 no function		
2	0.097 no function		0.8 testvolume							
3				0.8 listdir basic mkdir mkfile NewShortName readfile renamefile renamefile readfile NewShortName movefile mkfile mkdir listdir basic convfile						
4					0.8 testvolume SetDelParams readfile NewWildcardName NewDirName mkfile mkdir listdir basic InitialiseDelPa del BaseDir			0.195 no function		

	0	1	2	3	4	5	6	7	8	9
5						0.8 testvolume readfile NewWildcardName NewDirName mkfile mkdir listdir basic InitialiseCount BaseDir		0.097 no function		
6							0.8 renamefile readfile NewShortName mkfile mkdir listdir basic			
7		0.039 NewShortName			0.039 InitialiseDelPa	0.039 InitialiseCount		0.8 testvolume readfile NewWildcardName NewDirName mkfile mkdir listdir basic BaseDir		0.039 InitialiseSubDi
8									0.8 BaseDir listdir basic mkdir mkfile NewDirName NewWildcardName readfile testvolume	
9								0.195 no function		0.8 testvolume SetSubDir rmdir readfile NewWildcardName NewDirName mkfile mkdir listdir basic InitialiseSubDi BaseDir

	0	1	2	3	4	5	6	7	8	9
10			0.049 no function			0.049 BaseDir				
11	0.097 no function									
12	0.195 no function									
13									0.195 no function	
14									0.195 no function	
15									0.097 no function	

	0	1	2	3	4	5	6	7	8	9
16								0.065 no function	0.065 no function	
17	0.195 no function									
18		0.097 no function								
19										

	10	11	12	13	14	15	16	17	18	19
Params	TestVol1 InvFileCount1 FileCount1	InvFileCount1 FileCount1	SubDir1	TestVol1 FileName2 FileCount1 SubDir1	TestVol1 FileName2 FileCount1 DelParams1	TestVol1 FileName2 InvFileCount1 FileCount1	FileName1 TestVol1 FileName2 FileCount1	DelParams1	FileName1 TestVol1 ShortFileName1 FileName2 FileCount1	
0		0.249 InitialiseCount	0.249 InitialiseSubDi					0.249 InitialiseDelPa		0.005
1									0.097 BaseDir	0.005
2	0.097 InitialiseCount									0.005
3										0.005
4										0.005

	10	11	12	13	14	15	16	17	18	19
5	0.097 no_function									0.005
6										0.005
7							0.039 BaseDir			0.005
8				0.049 InitialiseSubDi	0.049 InitialiseDelPa	0.049 InitialiseCount	0.049 BaseDir			0.005
9										0.005

	10	11	12	13	14	15	16	17	18	19
10	0 8 testvolume InitialiseCount	0 049 no_function				0 049 BaseDir				0.005
11	0 097 testvolume	0 8 InitialiseCount								0.005
12			0 8 SetSubDir InitialiseSubDi							0.005
13				0 8 BaseDir listdir_basic mkdir mkfile NewDirName NewWildcardName readfile rmdir testvolume SetSubDir InitialiseSubDi						0.005
14					0 8 BaseDir del listdir_basic mkdir mkfile NewDirName NewWildcardName readfile testvolume SetDelParams InitialiseDelPa					0.005
15	0 097 no_function					0 8 BaseDir listdir_basic mkdir mkfile NewDirName NewWildcardName readfile testvolume InitialiseCount				0.005

	10	11	12	13	14	15	16	17	18	19
16							0.8 BaseDir listdir_basic mkdir mkfile NewDirName NewWildcardName readfile testvolume readfile NewWildcardName NewDirName movefile mkfile mkdir listdir_basic convfile BaseDir		0.065 NewShortName NewShortName	0.005
17								0.8 SetDelParams InitialiseDelPa		0.005
18							0.097 no_function no_function		0.8 BaseDir listdir_basic mkdir mkfile NewDirName NewShortName NewWildcardName readfile renamefile testvolume renamefile readfile NewWildcardName NewShortName NewDirName movefile mkfile mkdir listdir_basic convfile BaseDir	0.005
19										1

APPENDIX B BUG DESCRIPTIONS

Bug 1. "error codes from the SUT do not match those from the oracle"

Severity	High
Number of Interface Files	0
Number of tests generated	0
Number of tests executed	0
First test number bug occurred on	All tests
Products containing bug	ELM 2.03.45i
Date bug found	Unknown
Configuration	
All configurations	
Interface File Summary	
Unrecorded	

Many of the test inputs into file system operations are invalid. It is expected that the file system will fail the attempted operation. However, for almost all failures the error code returned by Manager's file system often differs from the code returned by NTFS.

This could be a serious issue for OEM customers who write software to use Manager as a back-end to their solutions. They expect that Manager's file systems will return the same errors as given by the NTFS file systems. OEMs will often write their products using NTFS and then replace NTFS with Manager.

For the purposes of testing, this issue has been ignored by using rules to replace error messages by a single error message. This means that as long as an action fails (or succeeds) on both the SUT and oracle, the test does not fail.

It is extremely unlikely that this issue would have been identified using alternative testing techniques, unless it was manually identified as a potential issue and explicitly investigated.

Bug 2. "rd /s /q fails to remove files with very long names"

Severity	High
Number of Interface Files	9
Number of tests generated	100
Number of tests executed	100
First test number bug occurred on	Clean-up code
Products containing bug	ELM 2.03.20 (Development build) ELM 2.03.45i
Date bug found	28th September 2001
Configuration	
Tests run from Constance. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewDirName(inout string FileName, inout integer FileCount) mkfile(in string FileName) mkdir(in string FileName) InitialiseCount(out integer InvFileCount, out integer FileCount) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

This problem was found with the clean-up code run between tests (rather than with a specific generated test). The clean-up code used rd /s /q to recursively delete directories created during the previous test. The bug was found by the clean-up code, because the clean-up was more advanced (at this stage) than the operations being used for tests. This bug became apparent quite quickly during tests. It was not necessary to run the comparison.

This is an issue that had not been found during normal testing. The author suspected that there were issues with long name mangling in Manager. Filename mangling occurs when a long filename is converted into a short filename (eight characters, a dot and a three character extension). The short filename is accessible to DOS applications. However, problems with filename mangling had never been

seen before. It is unlikely this bug would have been found manually, unless the suspicions about filename mangling had been followed up.

Detection of this problem illustrates the importance of randomised testing. It is the way that the filenames were generated from the interface by the Alltest software that enabled this problem to be found.

Bug 3. "mkdir fails to make intermediate directories"

Severity	Medium
Number of Interface Files	9
Number of tests generated	100
Number of tests executed	100
First test number bug occurred on	1 (indexed from 0)
Products containing bug	ELM 2.03.20 (development build) ELM2.03.45i
Date bug found	28th September 2001
Configuration	
Tests run from Constance. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewDirName(inout string FileName, inout integer FileCount) mkfile(in string FileName) mkdir(in string FileName) InitialiseCount(out integer InvFileCount, out integer FileCount) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

If mkdir is used to create a directory where intermediate directories do not exist, then mkdir will not create the intermediate directories on Manager. However, on NTFS file systems mkdir does create intermediate directories if they do not exist. Prior to Alltest discovering this problem the author was not aware that mkdir could create intermediate directories if they did not exist. Having "extended attributes" enabled on the machine that the tests are being run from may cause the issue, as extended attributes do change the behaviour of mkdir.

This is a real issue with Manager that was not detected previously because the behaviour of mkdir was not known. The combination of automatic test generation with predecessor software as an oracle has enabled this bug to be discovered.

Bug 4. "mcf_open hangs when drive is mapped across a network"

Severity	High
Number of Interface Files	9
Number of tests generated	100
Number of tests executed	45
First test number bug occurred on	Clean-up code before test number 45
Products containing bug	ELM 2.03.45i
Date bug found	23rd November 2001
Configuration	
Tests run from Indefatigable. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewDirName(inout string FileName, inout integer FileCount) mkfile(in string FileName) mkdir(in string FileName) InitialiseCount(out integer InvFileCount, out integer FileCount) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

When running the tests, they will sometimes stop during clean-up. Leaving the test to run for 1/2 hour or more does not make any difference.

Analysis of the problem revealed that as part of the clean-up routine mcf_open is called, but never returns. The problem may be related to sharing Manager's drive letter over a network. The machine running the tests then maps to this network drive. A workaround for the problem is to un-map the drive before clean-up and re-map it after clean-up.

This is an issue with Manager that has not been seen before and could affect many customers. It was found by clean-up code, as the clean-up code is more advanced than the tests currently being executed. It took a while to isolate the issue and find the cause. Once the cause was found it was possible to provide a work around. This issue prevented further testing, so it was imperative to isolate it. It is

extremely unlikely that this issue would have been found with manual testing, as the repetitiveness of the automated tests almost certainly contributed to the issue being found.

Bug 5. "creating a new file failed"

Severity	High
Number of Interface Files	13
Number of tests generated	10
Number of tests executed	10
First test number bug occurred on	1 (Index from 0)
Products containing bug	ELM 2.03.45i
Date bug found	29th November 2001
Configuration	
Tests run from Indefatigable. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) renamefile(in string ShortFileName, in string FileName) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewShortName(out string ShortFileName, in string FileName) NewDirName(inout string FileName, inout integer FileCount) movefile(in string FileName2, in string FileName1) mkfile(in string FileName) mkdir(in string FileName) listdir_basic(in string FileName) InitialiseCount(out integer InvFileCount, out integer FileCount) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

Manager failed with the following:

```
Making File t:/File1
FS_OpenFile t:/File1 w
FS_OpenFile: couldn't open "t:/File1": invalid argument
Unable to create t:/File1
```

The file creation should have succeeded, as creating the file on the oracle was successful.

This bug took a lot of investigation to narrow down the scenario causing the problem. The first question was: is this issue random or reproducible? The

problem occurred on a number of tests meaning that it was repeatable, though not reproducible because it appeared to be random.

The first step was to eliminate possible external causes. Different hardware was tried (drive, media and PC were all changed) and the problem still occurred. The next step was to eliminate the possibility of the problem occurring within the test suite software (Alltest or Tcl). Changes to the calls being made in Tcl were tried, especially the fileopen and fileclose calls, which were re-written as a C DLL instead of using the Tcl commands. The changes did not affect the bug. This made it unlikely that the problem was within Alltest. Further examination of the Tcl code itself made it unlikely that the problem was with Tcl.

This made it more likely that the cause of the bug was Manager. To finally rule out Alltest and Tcl, a system monitoring tool (filemon[1]) was used to show what calls were being sent to Manager and how Manager was responding. The author ran the tests on Manager, stopping them after 2 hours (test34) to see if sufficient information has been gathered. The comparison was run: 35 tests executed, 26 passed, 9 failed.

The failed tests were:

test1 - rename issue
test2 - rename issue
*test6 - creation of file issue (once)
test11 - move errors different
*test15 - creation of file issue (once)
test26 - move failure (new problem)
test28 - move failure (new problem)
*test29 - creation of file issue (once)
test34 - rename issue

Looking at the three tests which had failed with the file creation, and examining filemon's log file, it became clear that:

[1] Filemon is the monitoring tool used. It is a freeware tool available from http://www.sysinternals.com. It can be used to monitor the low-level system calls being made to the kernel level file system device driver.

1. Command sent to Manager was no different to those when manager succeeded to create the file.

2. The failure was always the first file access onto the Manager controlled media.

So, if the requests to create files were absolutely correct, and all the failures were occurring at the very first file access following the clean-up, the problem is definitely with Manager.

The clean-up code was modified to add a configurable 15 second delay. Re-running the tests resulted in no failures due to writes randomly failing.

To conclude, it is likely that the mcf_format command (which formats the media during clean-up) is returning before the media is fully mounted. Consequently, writes immediately after the format sometimes fail. This is a timing issue (confirmed when the 15 second delay prevented this problem from happening), but the problem lies entirely with Manager.

This bug took a lot of manual effort to isolate, once the problem was noted. It is likely that without automation the bug would have been ignored, or its cause wrongly attributed to hardware failure. However, the ability to re-run the tests repeatedly enabled the problem to be properly isolated.

Techniques that concentrate on unit or component testing would never have detected this bug. The problem was a timing issue with interaction between different functional areas. For example, component testing may have tested mcf_format (which is an administrative function) or tested writing data to a file through the file system interface. However, it is by combining these functions through system level testing that this bug was found. This highlights the importance of system level testing. By using an oracle aimed at system level testing this bug was detectable using automated testing.

Bug 6. "rename file over itself has different result on SUT"

Severity	Low
Number of Interface Files	13
Number of tests generated	100
Number of tests executed	61
First test number bug occurred on	1 (Index from 0)
Products containing bug	ELM 2.03.45i
Date bug found	19th February 2002
Configuration	
Tests run from Constance. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) renamefile(in string ShortFileName, in string FileName) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewShortName(out string ShortFileName, in string FileName) NewDirName(inout string FileName, inout integer FileCount) movefile(in string FileName2, in string FileName1) mkfile(in string FileName) mkdir(in string FileName) listdir_basic(in string FileName) InitialiseCount(out integer InvFileCount, out integer FileCount) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

The following command copies a file over itself:

```
Rename fileX fileX
```

On NTFS there is no error, however on Manager the rename failed:

```
A duplicate file name exists, or the file cannot be found.
```

This is not a severe error. However, if Manager is to behave in the same way as NTFS then this error should not have occurred. Again discovery of this bug highlights the importance of an appropriate automated oracle for use in component or system testing. Without such an oracle this bug would not have been detected.

Bug 7. "net use /d hangs"

Severity	High
Number of Interface Files	13
Number of tests generated	100
Number of tests executed	4
First test number bug occurred on	Clean-up code before test number 4
Products containing bug	ELM 2.03.45i
Date bug found	19th February 2002
Configuration	
Tests run from Constance. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) renamefile(in string ShortFileName, in string FileName) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewShortName(out string ShortFileName, in string FileName) NewDirName(inout string FileName, inout integer FileCount) movefile(in string FileName2, in string FileName1) mkfile(in string FileName) mkdir(in string FileName) listdir_basic(in string FileName) InitialiseCount(out integer InvFileCount, out integer FileCount) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

net use /d hangs during clean-up code. This occurs when the previous test was very short. It seems likely that the problem is a timing issue within Manager. A five second pause was added to the beginning of the clean-up code. When the tests were rerun they all ran successfully without the clean-up hanging.

This is the fourth issue that has been found because of the way the clean-up code behaves. Again this is an interaction of the clean-up code with the type of tests that have been run. Running the clean-up code alone would not have found the bug.

Bug 8. "move directory fails with access is denied"

Severity	High
Number of Interface Files	13
Number of tests generated	100
Number of tests executed	100
First test number bug occurred on	Test number 26 (Index from 0)
Products containing bug	ELM 2.03.45i
Date bug found	26th June 2002
Configuration	
Tests run from Constance. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) renamefile(in string ShortFileName, in string FileName) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewShortName(out string ShortFileName, in string FileName) NewDirName(inout string FileName, inout integer FileCount) movefile(in string FileName2, in string FileName1) mkfile(in string FileName) mkdir(in string FileName) listdir_basic(in string FileName) InitialiseCount(out integer InvFileCount, out integer FileCount) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

Create a directory then attempt to move it. This results in an "access is denied" error being returned. The directory is empty (no files have been created in it).

Create a directory:	`t:\file7\ede.2`
Attempt to move (empty) directory:	`move t:\file7\ede.2 file10`
Receive error:	`access is denied`

The issue can be reproduced at the command line. This is a bug in Manager, though investigation is needed to locate the cause of the problem.

This bug should have been identified with good quality module testing. It is surprising to find a bug that can so easily be reproduced in software that has undergone thorough manual testing and many years of use by various customers.

Bug 9. "renaming files produces empty file"

Severity	Very high (causes data corruption)
Number of Interface Files	19
Number of tests generated	25
Number of tests executed	25
First test number bug occurred on	Test 4 (index from 0)
Products containing bug	ELM 2.03.45i
Date bug found	12th July 2002
Configuration	
Tests run from Constance. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) SetSubDir(inout integer SubDir) SetDelParams(inout integer DelParams) rmdir(in integer SubDir, in string FileName) renamefile(in string ShortFileName, in string FileName) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewShortName(out string ShortFileName, in string FileName) NewDirName(inout string FileName, inout integer FileCount) movefile(in string FileName2, in string FileName1) mkfile(in string FileName) mkdir(in string FileName) listdir_basic(in string FileName) InitialiseSubDir(out integer SubDir) InitialiseDelParam(out integer DelParams) InitialiseCount(out integer InvFileCount, out integer FileCount) del(in integer DelParams, in string FileName) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

The following steps reproduce this issue:

create 2 files.
Read second file created
Rename second file over first file (will fail)
Rename first file to a new filename
read new file - blank bytes returned!

This problem cannot be reproduced from the command line. However, the isolated sequence of steps is:

`write File4`	
`write File9`	
`read File9`	
`rename File9 File4`	This will fail as file4 exists
`rename File4 File7`	Successful
`read File7`	This will read blank bytes, not the data originally written in File4

At the end of this sequence of steps File4 still exists but it has a size of zero bytes. Both File9 and File7 also exist, and they have the correct number of bytes.

This issue is reproducible, though further investigation is needed to identify where in Manager the problem is occurring.

Detection of this bug highlights the importance of stochastic testing. It is extremely unlikely that a test engineer would have written a test which included these steps manually. Additionally, once the steps to cause the bug were isolated it became clear that the bug could not be reproduced from the command-line, a script was needed to carry out the test. This reinforces the importance of using scripts to carry out testing tasks, as the time taken between steps is often critical.

Bug 10. "rename file21 *1.* has in different message on SUT"

Severity	Low
Number of Interface Files	19
Number of tests generated	300
Number of tests executed	300
First test number bug occurred on	Test number 14 (Index from 0)
Products containing bug	ELM 2.03.45i
Date bug found	2nd August 2002
Configuration	
Tests run from Indefatigable. Manager running on Aquarius controlling a stand-alone drive. Disk in Plasmon format.	
Interface File Summary	
testvolume(out string TestVol) SetSubDir(inout integer SubDir) SetDelParams(inout integer DelParams) rmdir(in integer SubDir, in string FileName) renamefile(in string ShortFileName, in string FileName) readfile(in string FileName) NewWildcardName(inout string FileName, inout integer FileCount) NewShortName(out string ShortFileName, in string FileName) NewDirName(inout string FileName, inout integer FileCount) movefile(in string FileName2, in string FileName1) mkfile(in string FileName) mkdir(in string FileName) listdir_basic(in string FileName) InitialiseSubDir(out integer SubDir) InitialiseDelParam(out integer DelParams) InitialiseCount(out integer InvFileCount, out integer FileCount) del(in integer DelParams, in string FileName) copyfile(in string FileName2, in string FileName1) BaseDir(out string FileName, in string TestVol, inout integer FileCount)	

Rename a file using a wild card. The message returned by Manager is different from that returned by NTFS.

APPENDIX C TEST AND COMPARE (TAC) DATA FILE USING PATTERNS

This appendix details the Test and Compare (TAC) data file.

For every command (or set of commands) that need to be run, a data file is created. This has the following form:

```
test_command "dir *.* /b /s"
BEGIN
MANYNOORDER testdirectory;Exact(fn)
END
```

Test commands are placed at the beginning of the file. These are the commands that form the test. Each test command line starts with the keyword `test_command`. If multiple `test_command` lines are given, then each command will be executed in turn.

The following key words can be used at the beginning of a test file:

`test_command`	specifies the command(s) to execute from the command line for this test
`test_file`	specifies a Tcl file that contains the tests. If this is specified then any `test_command` key words will be ignored.

If lines do not begin with any of the keywords specified above then they specify new patterns for matching data against, or for redefining the existing patterns used.

BEGIN marks the start of the actual patterns used for matching against the data. The following describes the keywords that will found at the beginning of the lines following this:

BEGIN/END	defines the beginning and end of the pattern section
MANY	specifies that this pattern should match one or more lines
MANYNOORDER	specifies that this pattern matches one or more lines. But that the output from the SUT and the oracle may be in different orders. Puts the lines into alphabetical order before comparing, ensuring that they should match.
ONCE	this pattern will only match one line.

After END, no more is read from this data file. Therefore, if any comments are needed, they can go at the bottom of a data file. It is very useful to put comments into a test file. They ensure that maintenance in the future is as easy as possible.

Patterns consist of descriptive names, and actions to carry out. The following are defined:

date	has the form DD/MM/YY or D/M/Y or a combination
time	has the form hh:mm or h:m or a combination
int	any integer number
intthous	any integer number possibly containing thousand separators: 10,000 1,234 123 etc.
decimal	any number possibly containing a decimal place: 10.001 1.234 123 etc.
fn	filename of the form abcdef.xxx or .123 or abcdef etc.
anything	matches anything on the line. Careful use ought to be made of this. DO NOT use at the beginnings of lines where proper matches need to be made. It can be used at the end of lines where the contents really do not matter
blank	an entirely blank line, use this pattern on it's own.
ws	whitespace. There is almost certainly no need to use this. The pattern matching allows for whitespace between variables defined, though they will still match even if there is no white space.

Additional patterns can be set up if needed, or if patterns need to be redefined for a particular test definition. This is done at the beginning of the data file. The pattern matching is done using regular expressions as implemented in Tcl. For example, the definition for date is: `[0-9]?[0-9]/[0-9]?[0-9]/[0-9]?[0-9]`. This will match a string such as `01/07/99`. It will also match a string like `99/98/97`. This is

not a valid date. To create a tighter definition for date, the following line is included at the top of the data file (before BEGIN):

```
date {[0-3]?[0-9]/[0-1]?[1-9]/[0-9][0-9]}
```

This definition will then be used in place of the default definition. This is useful for readability in a script, where there is no need to have variables called "mydate" or "date1" etc.

In the same way, new definitions can be made, specifying a name that is not a keyword or a current pattern variable.

Finally, there are the comparison keys. These do not define the pattern, but how the pattern should be treated to confirm that the match is correct. If a pattern variable is used on it's own, then no further checking is made, the match of the string to the pattern is sufficient, and the SUT does not need to match the oracle.

Alternatively, a pattern can be the parameter for a comparison key. The comparison keys currently defined are:

`Range`	specifies that the matched number (it has to be a number nothing else will make sense) should be within a given range. There are three parameters:
	`Range(pattern, x, y)`
	`Pattern` the name of the pattern variable
	`X` the lower range bound
	`Y` the upper range bound
	if sutmatch is the number matched on the SUT and ormatch is the number matched on the oracle then:
	lower bound = ormatch - x
	upper bound = ormatch + y
	lower bound <= sutmatch <= upper bound
	If the pattern fails this match (including a check for none numeric match) then the whole test will fail at this point.

`exact`	the match from the SUT must be identical to the oracle: `Exact(pattern)`
`literal`	this comparison key contains a literal string rather than a pattern: `Literal("Hello this is a literal string")`

oneof	instead of a single match, the pattern can match one of a number of possibilities. This is the only comparison key that can contain other comparison keys. However, oneof cannot contain another oneof, because it is unnecessary. Each parameter in oneof is separated by "|":
	`Oneof(pattern|Range(pattern,x,y)|Literal("xx"))`
testdirectory	this is treated in the same way as a pattern keyword. However, it cannot be redefined. It specifies that the pattern should match the path of the system that the test is currently being run on.

Each comparison key or pattern variable on the line is separated by ";". This specifies that the patterns to be matched are separated by zero or more white space. White space consists of tabs and spaces. New lines are not included, as each line in the data file should match a whole line of test output.

There are occasions when a pattern is used so often, that it needs to be defined and used when needed. What is needed is a "procedure" specifying a common pattern. This is achieved using USE and SUB:

```
test_command dir
test_numfiles 10
test_dirdepth 2

USE library/filesys.lib

BEGIN
SUB directory_and_dots
END
```

Where the file library/filesys.lib contains:

```
DEFSUB directory_and_dots
BEGIN
ONCE Literal(Volume in drive);anything
ONCE Literal(Volume Serial Number is);anything
ONCE blank
ONCE Literal(Directory of);anything
ONCE blank
ONCE date;time;Literal(<DIR>);Literal(.)
ONCE date;time;Literal(<DIR>);Literal(..)
MANYNOORDER
Exact(date);Exact(time);oneof(Literal(<DIR>)|Range(intthous,5,5));Exact(fn)
ONCE Exact(int);Literal(File(s));Range(intthous,5,5);Literal(bytes)
ONCE intthous;Literal(bytes free)
SUB END
```

Library files can contain multiple patterns. A test file can specify multiple files to use patterns from.

There are a number of things that are not covered:

1. Sub patterns cannot use other sub patterns. Though this is quite a limitation, this "language" is intended for quite a restricted purpose, therefore making this limitation less of a problem.

2. Sub patterns cannot be declared in a main test file, they must be declared in a separate file and then references using the "USE" keyword.

3. There is no way to specify that a sub pattern may be repeated multiple times. A "MANYSUB" keyword as required.

To run the tests a customised Tcl interpreter runs a script called `TAC.TCL`. The data file is interpreted by TAC, which in turn runs a series of scripts interpreted by Tcl.

APPENDIX D GENERATING THE TRANSITION MATRIX

Chapter 2 section 2.3.3 describes two algorithms for generating the states in the Markov Chain Transition Matrix. This appendix presents other approaches that were tried. Each section describes the problems with the matrix generated.

Each of the following methods is illustrated with five functions:

```
One (out X, inout Y)
Two (in X, out Z)
Three (out Y, out W)
Four (in W)
Five (in Y, in Z)
```

Method Attempt 1

The first approach is to consider states based upon inputs (and combinations of those inputs) to functions. This produces the following list of inputs:

```
Y, X, W, YZ
```

Taking these inputs this produces the following states:

```
Y = Y
X = X
W = W
YZ = Y, Z, YZ
```

Summary of states:

```
Y, X, W, Z, YZ
```

This will not produce a usable matrix. For instance Parameters Y and W are only ever initialised together, yet these never appear together in the matrix, so function Three will not appear in the matrix so cannot be called. The matrix is not shown for this method – as it is trivial. See the later methods for example of how a matrix looks.

Method Attempt 2

The next approach is to use combinations of outputs with inputs.

The inputs are:

```
Y, Z, W, YZ
```

The outputs are:

```
XY, Z, YW
```

The states produced from these are:

```
Y, Z, W, YZ, X, XY, YW
```

Giving the following matrix:

	0	Y	Z	W	YZ	X	XY	YW
0								Three
Y							One	Three
Z								
W				Four				Three
YZ					Five			
X								
XY		unset						
YW	unset	unset		unset				Three

Note that the state XZ is not in the matrix. Therefore function Two is not in the matrix and cannot be run. This is because the inputs and outputs have been handled separately.

All other proposed methods attempt to combine the inputs and outputs.

Method Attempt 3

States are generated in the following manner:

1. Examine the output parameters. Make a list of output parameters (ignore parameters that act as inputs and outputs) that only appear in groups. These groups can then be considered as a single parameter.

2. Produce combinations of inputs, ensuring that the special combined parameters are always used together.

Output parameters are:

```
X, Z, YW
```

Input parameters are:

```
Y, X, W, YZ
```

Expanding the combinations of the input parameters gives:

```
Y, X, W, Z, YZ
```

Replacing all instances of W and Y with YW (from the outputs) gives the final list of states:

```
WY, X, Z, WYZ
```

Note the output parameters should also be included in the list. This is because an output may not be used in any functions as an input. However, this example uses all the set outputs in at least one function as inputs.

The state XZ is not present. The matrix produced using this method is not usable.

Method Attempt 4

The next step is to consider not just the inputs to functions, but all the parameters used or set by a function. The result is the following algorithm:

1. Go through the functions and build a Variable Table (VT). The VT contains a list of all output parameters. It is indexed on each output parameter. For each output parameter, there is a list of all output parameters it is used with. For example, if we have 4 functions that initialise parameters A, AB, BC, C. Then the VT will be: A = {A, AB}; B = {AB, BC}; C= {BC, C}.

2. For each of the functions, create a list of all parameters.

3. Create combinations of these parameters to produce unique states.

4. Expand each state with the grouping information in the VT.

Using the same example as before:

Step 1, build the VT:

```
X = {XY}; Y = {YW, XY}; Z = {Z}; W = {YW}
```

Step 2, list the parameters for each function:

```
XY, XZ, YW, W, YZ
```

Step 3, create combinations of these parameters.

```
XY = {X, Y, XY}
XZ = {X, Z, XZ}
YW = {Y, W, YW}
W = {W}
YZ = {Y, Z, YZ}
```

Summarising:

```
X, Y, XY, Z, XZ, W, YW, YZ
```

Step 4, Expand with information from VT

```
X = XY
Y = YW, XY
XY = XYWY, XYXY = XYW, XY
```

```
Z = Z
XZ = XYZ
W = YW
YW = YWYW, XYYW
ZY = ZYW, ZXY
```

Removing duplicates gives the following states:

```
XY, YW, XYW, Z, XYZ, YWZ
```

Immediately this is better, since the state XYZ means that function Two can be called. This produces the following matrix:

	0	XY	YW	XYW	Z	XYZ	YWZ
0			Three				
XY		One		Three		Two	
YW	Unset		Three, Four	One			
XYW		Unset	Unset	One, Three, Four			
Z							Three
XYZ		Unset				One, Two, Five	
YWZ					Unset		Three, Four, Five

All functions have been included in this matrix. It is necessary to see if the matrix is commutable. For this the checking algorithm is used (see Appendix E).

States	State Used	State Reached
XY	Yes	II
YW	Yes	II
XYW	Yes	II
Z		
XYZ	Yes	I
YWZ		

State Z (and consequently YWZ) cannot be reached. It is possible that an Unset function could be added to get to Z from XYZ. However, there is no function to get from Z straight back to XYZ, so this is a dangerous solution.

APPENDIX E CHECKING THE MATRIX

A further algorithm was developed to confirm that the matrix created contained states that could always be transitioned into and out of. This was used manually and also implemented into Alltest.

Create a table with three columns. In the first column list all the states in the matrix excluding the initial state and the end state. The second column lists the states that are reachable. The third column lists the states that have been checked.

Examine the matrix starting at the initial state. In the second column mark those states that can be reached from the initial state.

Now, loop though the table:

- Select a state that is reachable (this is marked in the second column), but has not been checked (this is marked in the third column). The state selected is the "current state".
- Examine the transition matrix and mark in the second column the states that are reachable from the "current state".
- Mark in the third column that the "current state" has been checked.

Continue until all states that are reachable have been checked.

Once the checking has stopped, examine the table for any states that have not been reached. If any exist, the matrix is not fully commutable.

APPENDIX F TESTING THE GENERATOR

This appendix lists the data used to generate the charts in Chapter 9. The charts produced are Figure 29 and Figure 30.

Data for Changing Probability of Staying in Same State

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.001	0.001	**Test 1**				
		Transitions	1	6940	1039.9	1037.7
		Same States	0	13	0.9	1.4
		Change States	1	6936	1038.9	1036.7
		Action Calls	0	5	0.3	0.7
		Init Calls	0	3471	520.6	519.0
		Unset Calls	1	3467	518.9	518.3
0.001	0.050	**Test 2**				
		Transitions	1	6526	1066.4	1024.6
		Same States	0	317	47.8	46.4
		Change States	1	6261	1018.6	978.7
		Action Calls	0	105	16.7	16.7
		Init Calls	0	3299	541.0	519.1
		Unset Calls	1	3131	508.8	489.3
0.001	0.100	**Test 3**				
		Transitions	2	6514	1025.8	1029.2
		Same States	0	592	92.5	94.2
		Change States	2	5922	933.3	935.5
		Action Calls	0	225	32.0	33.6
		Init Calls	1	3329	527.7	528.6
		Unset Calls	1	2960	466.1	467.8
0.001	0.150	**Test 4**				
		Transitions	2	8414	949.7	947.7
		Same States	0	1100	128.6	129.0
		Change States	2	7314	821.1	819.2
		Action Calls	0	404	45.0	46.5
		Init Calls	1	4354	494.7	492.4
		Unset Calls	1	3656	410.0	409.6

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.001	0.200	**Test 5**				
		Transitions	1	7445	995.3	980.7
		Same States	0	1330	180.6	179.5
		Change States	1	6115	814.6	801.8
		Action Calls	0	497	62.8	63.8
		Init Calls	0	3891	525.7	517.0
		Unset Calls	1	3057	406.8	400.9
0.001	0.250	**Test 6**				
		Transitions	3	6742	1071.1	1041.1
		Same States	0	1500	245.0	239.7
		Change States	3	5242	826.1	802.0
		Action Calls	0	551	85.3	85.6
		Init Calls	2	3598	573.4	555.7
		Unset Calls	1	2620	412.4	401.0
0.001	0.300	**Test 7**				
		Transitions	1	8554	1083.8	1081.4
		Same States	0	2337	299.0	301.0
		Change States	1	6217	784.8	781.1
		Action Calls	0	819	104.5	108.6
		Init Calls	0	4627	587.5	583.7
		Unset Calls	1	3108	391.8	390.5
0.001	0.350	**Test 8**				
		Transitions	2	6825	1011.0	1054.3
		Same States	0	2220	327.5	343.7
		Change States	2	4605	683.5	711.3
		Action Calls	0	804	113.6	122.1
		Init Calls	1	3719	556.2	578.1
		Unset Calls	1	2302	341.2	355.6
0.001	0.400	**Test 9**				
		Transitions	1	6970	1049.0	1038.1
		Same States	0	2600	390.8	389.3
		Change States	1	4370	658.1	649.5
		Action Calls	0	896	135.5	138.6
		Init Calls	0	3931	585.0	576.6
		Unset Calls	1	2185	328.5	324.8

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.001	0.450	**Test 10**				
		Transitions	1	6285	1034.4	1006.1
		Same States	0	2651	435.7	427.0
		Change States	1	3634	598.7	579.7
		Action Calls	0	957	151.3	153.3
		Init Calls	0	3594	584.4	565.0
		Unset Calls	1	1817	298.8	289.9
0.001	0.500	**Test 11**				
		Transitions	1	9202	997.3	995.0
		Same States	0	4449	468.3	471.7
		Change States	1	4753	529.0	524.2
		Action Calls	0	1632	162.2	169.4
		Init Calls	0	5194	571.1	566.1
		Unset Calls	1	2376	264.0	262.0
0.001	0.550	**Test 12**				
		Transitions	1	9917	1044.3	1049.1
		Same States	0	5106	542.7	547.7
		Change States	1	4811	501.6	502.1
		Action Calls	0	1875	188.7	197.7
		Init Calls	0	5638	605.4	603.4
		Unset Calls	1	2404	250.2	251.0
0.001	0.600	**Test 13**				
		Transitions	2	8033	957.3	968.4
		Same States	0	4603	544.9	555.6
		Change States	2	3430	412.4	413.5
		Action Calls	0	1590	187.0	196.9
		Init Calls	1	4729	564.6	568.3
		Unset Calls	1	1714	205.6	206.7
0.001	0.650	**Test 14**				
		Transitions	1	7774	1005.7	985.3
		Same States	0	4790	624.6	615.4
		Change States	1	2984	381.1	370.7
		Action Calls	0	1652	214.7	219.3
		Init Calls	0	4631	601.0	584.8
		Unset Calls	1	1491	189.9	185.3

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.001	0.700	**Test 15**				
		Transitions	1	7540	1018.4	1031.7
		Same States	0	5041	684.6	697.7
		Change States	1	2499	333.9	334.8
		Action Calls	0	1671	232.8	246.8
		Init Calls	0	4619	619.3	622.5
		Unset Calls	1	1250	166.3	167.4
0.001	0.750	**Test 16**				
		Transitions	2	7895	1075.6	1105.5
		Same States	0	5704	779.9	805.3
		Change States	2	2191	295.8	301.0
		Action Calls	0	1939	262.8	284.1
		Init Calls	1	5007	665.6	677.3
		Unset Calls	1	1095	147.3	150.5
0.001	0.800	**Test 17**				
		Transitions	3	8473	1015.0	1002.4
		Same States	0	6608	789.4	782.9
		Change States	2	1865	225.6	220.3
		Action Calls	0	2275	266.0	278.9
		Init Calls	2	5510	636.8	622.0
		Unset Calls	1	932	112.2	110.1
0.001	0.850	**Test 18**				
		Transitions	1	9160	1099.7	1070.3
		Same States	0	7569	914.4	894.7
		Change States	1	1591	185.3	176.5
		Action Calls	0	2760	305.0	323.0
		Init Calls	0	5605	702.6	671.4
		Unset Calls	1	795	92.0	88.2
0.001	0.900	**Test 19**				
		Transitions	2	7895	1031.7	1069.3
		Same States	0	7055	913.7	951.7
		Change States	2	840	118.0	118.5
		Action Calls	0	2493	293.7	344.9
		Init Calls	1	4983	679.6	681.7
		Unset Calls	1	419	58.4	59.2

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.001	0.950	**Test 20**				
		Transitions	3	8993	1080.3	1088.8
		Same States	1	8489	1016.7	1028.6
		Change States	2	504	63.6	61.2
		Action Calls	0	3527	305.4	381.4
		Init Calls	2	5904	743.7	714.0
		Unset Calls	1	251	31.2	30.5
0.001	0.995	**Test 21**				
		Transitions	1	6347	987.8	941.0
		Same States	0	6317	980.1	936.1
		Change States	1	32	7.6	5.8
		Action Calls	0	2230	78.7	228.9
		Init Calls	0	6332	905.5	823.5
		Unset Calls	1	16	3.6	2.8

Data for Changing Probability of Ending Test

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.001	0.001	**Test 1**				
		Transitions	1	7310	1056.1	1038.9
		Same States	0	9	0.9	1.3
		Change States	1	7301	1055.2	1038.0
		Action Calls	0	5	0.3	0.7
		Init Calls	0	3656	528.7	519.6
		Unset Calls	1	3650	527.1	519.0
0.020	0.001	**Test 2**				
		Transitions	1	466	47.1	46.1
		Same States	0	2	0.0	0.2
		Change States	1	466	47.1	46.1
		Action Calls	0	1	0.0	0.1
		Init Calls	0	234	23.9	23.2
		Unset Calls	1	232	23.2	22.9
0.040	0.001	**Test 3**				
		Transitions	1	154	23.7	23.0
		Same States	0	1	0.0	0.1
		Change States	1	154	23.7	23.0
		Action Calls	0	1	0.0	0.1
		Init Calls	0	77	12.2	11.7
		Unset Calls	1	77	11.6	11.3
0.060	0.001	**Test 4**				
		Transitions	1	126	16.3	15.6
		Same States	0	2	0.0	0.1
		Change States	1	126	16.3	15.6
		Action Calls	0	1	0.0	0.1
		Init Calls	0	64	8.3	8.0
		Unset Calls	1	62	8.0	7.6
0.080	0.001	**Test 5**				
		Transitions	1	84	12.2	11.6
		Same States	0	1	0.0	0.1
		Change States	1	84	12.2	11.6
		Action Calls	0	1	0.0	0.0
		Init Calls	0	43	6.2	6.0
		Unset Calls	1	41	6.0	5.6

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.100	0.001	**Test 6**				
		Transitions	1	63	9.5	8.8
		Same States	0	1	0.0	0.1
		Change States	1	63	9.4	8.8
		Action Calls	0	1	0.0	0.0
		Init Calls	0	32	4.8	4.6
		Unset Calls	1	31	4.6	4.2
0.120	0.001	**Test 7**				
		Transitions	1	57	8.1	7.6
		Same States	0	1	0.0	0.1
		Change States	1	57	8.1	7.6
		Action Calls	0	0	0.0	0.0
		Init Calls	0	29	4.1	4.0
		Unset Calls	1	28	4.0	3.6
0.140	0.001	**Test 8**				
		Transitions	1	40	7.1	6.4
		Same States	0	1	0.0	0.1
		Change States	1	40	7.1	6.3
		Action Calls	0	0	0.0	0.0
		Init Calls	0	20	3.6	3.4
		Unset Calls	1	20	3.5	3.0
0.160	0.001	**Test 9**				
		Transitions	1	50	6.1	6.0
		Same States	0	1	0.0	0.1
		Change States	1	50	6.1	6.0
		Action Calls	0	1	0.0	0.0
		Init Calls	0	26	3.0	3.3
		Unset Calls	1	24	3.1	2.8
0.180	0.001	**Test 10**				
		Transitions	1	35	5.6	5.0
		Same States	0	1	0.0	0.1
		Change States	1	35	5.6	5.0
		Action Calls	0	0	0.0	0.0
		Init Calls	0	18	2.7	2.7
		Unset Calls	1	17	2.8	2.3

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.200	0.001	**Test 11**				
		Transitions	1	30	4.7	4.0
		Same States	0	1	0.0	0.1
		Change States	1	30	4.7	4.0
		Action Calls	0	0	0.0	0.0
		Init Calls	0	16	2.3	2.2
		Unset Calls	1	14	2.4	1.8
0.220	0.001	**Test 12**				
		Transitions	1	37	4.6	4.2
		Same States	0	1	0.0	0.0
		Change States	1	37	4.6	4.2
		Action Calls	0	0	0.0	0.0
		Init Calls	0	19	2.2	2.4
		Unset Calls	1	18	2.4	1.9
0.240	0.001	**Test 13**				
		Transitions	1	25	4.1	3.5
		Same States	0	1	0.0	0.0
		Change States	1	25	4.1	3.5
		Action Calls	0	0	0.0	0.0
		Init Calls	0	12	2.0	2.0
		Unset Calls	1	13	2.2	1.6
0.260	0.001	**Test 14**				
		Transitions	1	22	3.7	3.1
		Same States	0	2	0.0	0.1
		Change States	1	22	3.7	3.1
		Action Calls	0	0	0.0	0.0
		Init Calls	0	12	1.7	1.8
		Unset Calls	1	11	2.0	1.4
0.300	0.001	**Test 15**				
		Transitions	1	23	3.3	2.8
		Same States	0	0	0.0	0.0
		Change States	1	23	3.3	2.8
		Action Calls	0	0	0.0	0.0
		Init Calls	0	13	1.5	1.6
		Unset Calls	1	10	1.8	1.3

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.400	0.001	**Test 16**				
		Transitions	1	18	2.5	2.0
		Same States	0	0	0.0	0.0
		Change States	1	18	2.5	2.0
		Action Calls	0	0	0.0	0.0
		Init Calls	0	10	1.1	1.2
		Unset Calls	1	9	1.5	0.9
0.500	0.001	**Test 17**				
		Transitions	1	15	2.0	1.4
		Same States	0	0	0.0	0.0
		Change States	1	15	2.0	1.4
		Action Calls	0	0	0.0	0.0
		Init Calls	0	8	0.7	0.9
		Unset Calls	1	7	1.3	0.6
0.600	0.001	**Test 18**				
		Transitions	1	8	1.6	1.0
		Same States	0	0	0.0	0.0
		Change States	1	8	1.6	1.0
		Action Calls	0	0	0.0	0.0
		Init Calls	0	5	0.5	0.7
		Unset Calls	1	3	1.1	0.4
0.700	0.001	**Test 19**				
		Transitions	1	6	1.4	0.8
		Same States	0	0	0.0	0.0
		Change States	1	6	1.4	0.8
		Action Calls	0	0	0.0	0.0
		Init Calls	0	3	0.3	0.6
		Unset Calls	1	3	1.1	0.3
0.800	0.001	**Test 20**				
		Transitions	1	5	1.2	0.5
		Same States	0	0	0.0	0.0
		Change States	1	5	1.2	0.5
		Action Calls	0	0	0.0	0.0
		Init Calls	0	3	0.2	0.4
		Unset Calls	1	3	1.0	0.2

P(end)	P(same)		Minimum	Maximum	Average (mean)	Standard deviation
0.900	0.001	**Test 21**				
		Transitions	1	3	1.1	0.4
		Same States	0	0	0.0	0.0
		Change States	1	3	1.1	0.4
		Action Calls	0	0	0.0	0.0
		Init Calls	0	2	0.1	0.3
		Unset Calls	1	2	1.0	0.1
0.950	0.001	**Test 22**				
		Transitions	1	3	1.1	0.3
		Same States	0	1	0.0	0.0
		Change States	1	3	1.1	0.3
		Action Calls	0	0	0.0	0.0
		Init Calls	0	2	0.1	0.3
		Unset Calls	1	2	1.0	0.1

APPENDIX G AVERAGE LENGTH OF TEST

Chapter 3 gives the average length of the test as

$$t = \frac{1}{p}$$

Where: t is the average number of transitions in a test

p is probability of ending the test on this transition

The calculation for this is as follows:

q = probability of not ending the test on this transition

$q = 1 - p$

P_n = Probability of finishing on the nth step

$$t = \sum_{n=0}^{\infty} nP_n$$

$$= \sum_{n=0}^{\infty} n\left(q^{n-1}\right)p$$

$$= p\sum_{n=0}^{\infty} n\left(q^{n-1}\right)$$

$$= p\sum_{n=0}^{\infty} \frac{d}{dq}q^n$$

$$t = p\frac{d}{dq}\sum_{n=0}^{\infty} q^n$$

$$= p\frac{d}{dq}\left\{\frac{1-q^n}{1-q}\right\}$$

q^n tends to zero as n tends to infinity

$$t = p\frac{d}{dq}\left\{\frac{1}{1-q}\right\}$$

$$= p\left\{\frac{1}{(1-q)^2}\right\}$$

$$p = 1-q$$

$$\therefore$$

$$t = \frac{p}{p^2}$$

$$t = \frac{1}{p}$$

APPENDIX H USING ALLTEST

This appendix discusses how to debug the System Under Test (SUT) and how to analyse failed tests. It also describes how to install Alltest and how it is used to test Manager.

This appendix assumes that Alltest is run from the base directory, where the directories shown in Table 17 are sub directories from the base directory.

Directory	Description
Interface	Contains the TCD files making up the interface.
Library	Contains Tcl library files used by the interface.
Tests	The directory containing the generated test files.
Orclog	The directory where the oracle results reside. (configurable).
Sutlog	The directory where the SUT results reside. (configurable).
Results	The directory where the comparison writes the results from the test.
Rules	The directory containing global (supplied) rules for Alltest.
Compare	Contains the Tcl code for comparing the SUT and oracle results.

Table 17 Directory structure of Alltest

Debugging the Interface

If the rules have failed in some way, then they need to be updated. Care must be taken to ignore as little as possible. If rules are too general, then tests may start to pass when they should have failed. If there is test output that should be considered as equivalent, but the rules do not ignore the differences, then it is necessary to debug the rules. The tester needs to save the appropriate segments from the oracle and SUT output files into separate files and re-run the comparison for the single TCD file:

```
alltest k <tcd filename> <sut filename> <oracle filename> <result filename>
```

The tester should check that the results produced match those of the original comparison. If they do not the most likely cause is a difference in white-space as a result of copying the segments.

The next step is either to add further USERULE lines or to create new rules for the TCD file. The tester should keep re-checking the comparison until the results are correct.

The full comparison can then be carried out again on the whole test results. There is no need to re-run the tests on either system. Once the full comparison is complete, re-examine the results as before.

Debugging the System Under Test

If the problem appears to be with the SUT, there are a number of debugging strategies that can be employed. As with any development and debugging activity, the best approach will vary for different developers.

The first thing to do is to re-run the single test again and see if the results are the same. If the test did not fail in the same way, or did not fail at all, then it is likely that there is some timing bug that needs finding. Re-running part or all of the generated tests and seeing how many failures are different may help locate the problem. Timing issues are always difficult to fix. However having an automated tool that can re-run a set of tests identically can greatly reduce the frustration of identifying this type of issue.

If the test did fail in the same way, then it is likely to be a fully repeatable issue. The strategy to adopt here is to find the minimum number of steps needed to induce the problem. This will probably entail working backwards through the test output for the SUT and finding what steps were actually taken, and the values of parameters at the time of the failed action. It is useful to try and reproduce the issue manually. This is not always possible, especially with time critical problems. If the failure cannot be manually induced, then short test files can be written by hand for

Alltest to run. As the test files are actually Tcl, it is possible to set parameter values and force these to be inputs into TCD files.

Once failures have been analysed and their cause identified, then the SUT can be corrected and testing can continue.

Analysing Failed Tests

Once a test has been detected as failed, the test and the outcomes need to be analysed. Analysis is trying to establish:

- Is the failure repeatable, reproducible, or random? Re-running the test, and analysing other tests that have failed may help determine this.
- Did the SUT fail the test, or is this a false negative?
- If the SUT failed the test, which sequence of steps caused the failure? It may be possible to determine a set of steps smaller than the whole test.
- Can the failure be repeated manually or is timing critical so that the failure only occurs when the test is run from a script.

This investigation is important, as it is the first step in fixing the problem. It allows a thorough bug report to be written which a developer will use to discover where the software failed.

Examples of how this process is carried out can be seen in the bug reports in Appendix B.

Using Alltest to test Manager

Preparing and Installing Alltest to test Manager

The current implementation of Alltest includes a simple script for its set up and installation. To prepare an installation, compile and link alltest.exe and alltest.dll.

Then run: `tclsh83.exe prepare_setup.tcl` from the setup directory. This creates a directory (release) that contains all the components needed to install Alltest.

For testing Manager, Alltest is installed over two machines. The first machine is the test control machine. The second machine is the host machine that runs Manager.

On the test control machine, install the Tcl interpreter. Tcl832.exe which is in the release directory will install Tcl 8.3. Finish the Tcl installation by adding the Tcl\bin directory to the system path variable. Install Alltest by running setup.bat from the release directory.

On the host machine (which is running Manager), install Tcl8.3 as before. Also install the testtools module that contains the server and the Manger API. Compile the Manager API library (`mcf_server.dll`) against the current release of Manager.

Generating and Running the Tests

Edit `config.txt` (in `alltest` directory) to setup parameters ready for running the tests on the oracle. Figure 33 shows part of a configuration file. The file contains parameters that are used by Alltest during the generation and execution of tests. Figure 34 shows another part of the configuration file. This time the parameters are not understood by Alltest, but are used by the interface files and clean-up code to configure behaviour of the tests while running. Parameters in the configuration file allow the test behaviour to be modified. For example, a bug may be known, but it can be worked around until it is fixed. A parameter can be added to the config file, and the appropriate interface files altered so that the work around is used until the bug is fixed. This enables testing to continue without triggering the known bug.

```
# values used by alltest itself
markov_probability_end_test          0.005
markov_probability_same_state        0.8
generator_num_tests                  25

# files used to check the matrix the generator is producing
markov_log_filename                  output\matrix.txt
state_btree_log_filename             output\state.txt

# the value of the tcl interpreter
tcl_interp                           tclsh83

# path values used when running tests and comparing results
sut_jb_path                          q:/
orc_jb_path                          y:/
sut_hd_path                          c:/test1
orc_hd_path                          c:/test2

# the clean up script (tcl only)
test_cleanup                         cleanup_test.tcl
```

Figure 33 Config file showing parameters used by Alltest

```
# The rest of this file contains parameters that are defined by the
# interface files or clean-up code only.

# do the mkdir part with the bug intact
bug_mkdir_elements                   1
# do the movedirectory with the workaround for the bug
bug_movedir                          1

mgr_server_host                      aquarius
mgr_server_port                      2551
mgr_server_share                     standalone

# the cleanup code can either reformat media or
# it can delete directories, etc.
# if it is reformatting media, the following specify
# the filesystem:  plasmon, afs, udf
# and the volume name to use
cleanup_format_unit_num              1
cleanup_format_filesystem            plasmon
cleanup_format_name                  DISK001

# cleanup can pause at the beginning and end to help resolve timing issues
cleanup_start_pause                  5
cleanup_finish_pause                 15

# this specifies the cleanup_method:
#   deldir      recursively delete directories
#   format      reformat a stand-alone drive
#
# other methods can be added as required.
cleanup_method                       format
```

Figure 34 Config file showing parameters used by clean-up code and interface files

Generate the tests, from the command prompt:

```
cd alltest
alltest g .
```

Run the tests on the oracle system:

```
alltest r .
```

Rename `logs` directory to `orclog`. Re-edit `config.txt` to setup parameters ready for running the tests on the SUT. The parameters that most need changing will be the clean-up method as the method used on the SUT is different to the oracle, and the path on which the tests are being run.

```
alltest r .
```

Rename `logs` directory to `sutlog`. Re-edit `config.txt` to specify the correct parameters for the SUT and oracle. Run the comparison:

```
alltest c . .\sutlog .\orclog
```

Examine the results: `results\overall.log` gives the total number of tests that passed or failed, as well as the final result for each test. Examining individual logs will help determine the cause of the failure. For example, if `overall.log` gives test10 as failed, then examine `results\test10.log` to highlight the comparison that failed. Further re-runs of the test and investigation may then be needed to isolate the cause of the failure.

Setting up the Host Machine

On the host machine two processes need to be running: Manager and the server process. First configure the server process. Edit `server.ini` (in `testtools\tcl\utils\server`) to specify the port number used for communication between the server and the test controller. Run `tclsh83.exe mgr_server -i:server.ini`. Start running Manager. Share a disk or a drive from the jukebox so that the test controller can access the drive.

GLOSSARY

API Application Program Interface. This is a published interface to a product that allows developers to use the product within other applications.

C A compiled programming language.

CVS Concurrent Versions System. This is an open-source version control system.

Filemon This is a tool available from http://www.sysinternals.com. It is exceptionally useful when analysing problems on Manager. It consists of a file system filter driver and a console. The filter driver sits above any file system and monitors all the commands sent to the file system and the results of those commands. This information is printed out on the console. The data can be saved or automatically logged to a file for further analysis.

GUI Graphical User Interface.

Lex Lexical Analyser.

MCTM Markov Chain Transition Matrix

SUT System Under Test. This is the application or software being developed, and currently being tested.

TAC Test and Compare.

TCD Test Command Description.

Tcl Tool Command Language. This is an interpreted language. It is used extensively in the implementation of Alltest.

UOF Unit of Functionality.

VT Variable Table. This is a data structure created as part of the algorithm that generates the MCTM.

Yacc Yet Another Compiler-Compiler.

REFERENCES

Abramson, D., Sosic R. (1996). A Debugging and Testing Tool for Supporting Software Evolution. *Automated Software Engineering 3*, 1996, Pages 369 - 390.

Abramson, D., Watson, G. (2003) Debugging scientific applications in the .NET Framework. *Future Generation Computer Systems*, Vol 19, No 5, Pages 665-678.

Abramson, D., Watson, G., Dung, L. P. (2002) Guard: A tool for migrating Scientific Applications to the .NET Framework. *ICCS 2002, Lecture Notes in Computer Science*, Vol 2330, Pages 834-843.

Aichernig B. K. (1999) Automated black-box testing with abstract VDM oracles. *SAFECOMP '99, Lecture Notes in Computer Science*, Vol 1698, Pages 250-259.

Antoy S., Hamlet D. (2000) Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, Vol 26, No 1, Pages 55-69.

Armour, P. G. (2004) The Unconscious Art of Software Testing. *Communications of the ACM*, Vol 48, No 1, Pages 15-18.

Avritzer, A., Weyuker, E. J. (1995) The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, Vol 21, No 9, Pages 705-715.

Baresi L., Young M. (2001) Test Oracles. Technical Report CIS-TR-01-02. http://www.cs.uoregon.edu/~michal/pubs/oracles.html (accessed 4 September 2005)

BCS (2002) *A Glossary of Computing Terms, Tenth edition*. Addison Wesley. ISBN 0-201-77629-4.

Berman F., Fox G., Hey T. (2003), The Grid: Past, Present, Future. *Grid Computing: Making the Global Infrastructure a Reality*, Chapter 1, Pages 9-50, John Wiley and Sons, ISBN: 0470853190. Chapter 1 available from: http://www.grid2002.org/grid2002sample/chapter1.pdf (accessed 4 September 2005)

Bertolino, A. (2003) Software Testing Research and Practice, *ASM 2003, Lecture Notes in Computer Science*, Vol 2589, Pages 1-21.

Bertolino, A., Strigini L. (1996) On the use of testability measures for dependability assessment. *IEEE Transactions on Software Engineering*, Vol 22, No 2, Pages 97-108.

Beizer, B. (1990) *Software Testing Techniques, Second edition*. International Thompson Computer Press. ISBN 1-850-32880-3.

Beizer, B. (1997) Tutorial - An Overview of Testing. *International Software Quality Week Europe*, 4th November 1997.

Bicheno J. (1998) *The Quality 60*, PICSIE Books, ISBN 0 9513829 7 7.

Brealey S. (2000) University's £8m "Millennium Dome". *Varsity – The Cambridge Student Newspaper*, 29th September 2000.

Brilliant S. S., Knight J. C., Leveson N. G. (1990) Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, Vol 16, No 2, Pages 238-247.

Brown A. J. (1993) Specifications and Reverse-Engineering. *Software Maintenance: Research and Practice*, Vol. 5, Pages 147-153

Brown D. B., Roggio R. F., Cross II J. H. and McCreary C. L. (1992). An Automated Oracle for Software Testing. *IEEE Transactions on Reliability*, Vol. 41, No. 2, Pages 272 - 280.

Bullseye (2005) BullseyeCoverage Product Summary.
http://www.bullseye.com/productInfo.html (accessed 4 September 2005)

Cavarra, A., Crichton, C., Davies, J. (2004) A Method for the automatic generation of test suites from object models. *Information and Software Technology*, Vol 46, Pages 309-314

Climate Prediction (2005) ClimatePrediction.Net Gateway
http://www.climateprediction.net (accessed 23 October 2005)

Cohen D. M., Dalal S. R., Parelius J., Patton G. C. (1996) The Combinatorial Design approach to automatic test generation. *IEEE Software*, Vol 13, No 5, Pages 83-87

Cohen D. M., Dalal S. R., Fredman M. L., Patton G. C. (1997) The AETG System: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, Vol 23, No 7, Pages 437-444

Collins (1998) *Collins English Dictionary,* Fourth Edition, ISBN 0-00-472168-3.

Csallner, C., Smaragdakis, Y. (2004) JCrasher: an automatic robustness tester for Java. *Software – Practice and Experience*, Vol 34, No 11, Pages 1025-1050.

Cusumano M. A., Selby R. W. (1996) *Microsoft Secrets*, HarperCollins, 1996, ISBN: 0-00-255692-8

Cusumano, M., MacCormack, A., Kemerer, C. F., Crandall, B. (2003). Software Development Worldwide: The State of the Practice. *IEEE Software*, Vol 20, No 6, Pages 28-34.

Curtis H., Vella A., Burkhardt D., Broadbent M. (1998) Automated Test Suites from Reverse Engineering and Planguage; *Software Quality Week*, San Francisco, May 1998.

Dillon, E., Meudec, C. (2004) Automatic Test Data Generation from Embedded C Code. *SAFECOMP 2004, Lecture Notes in Computer Science*, Vol 3219, Pages 180-194.

Dustin E., Rashka J., Paul J. (1999) *Automated Software Testing*. Addison-Wesley, ISBN: 0-201-43287-0.

Edwards S. H. (2001) A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, Vol 11, No 2, Pages 97-111

Edwards, S. H., Sitaraman, M., Weide, B. W., Hollingsworth, J. (2004) Contract-Checking Wrappers for C++ Classes. *IEEE Transactions on Software Engineering*, Vol 30, No 11.

Finkelstein A. (2001) CAPSA and its implantation, Report to the Audit Committee and the Board of Scrutiny, University of Cambridge. Part A: Processes and decision-making. *Cambridge University Reporter*, Vol CXXXII No 6, 2nd November 2001, Pages 155-176

Finney K. (1996) Mathematical Notation in Formal Specification: Too Difficult for the Masses? *IEEE Transactions on Software Engineering*, Vol 22, No 2, Pages 158-159

Fox News (2003) Massive Blackout Cripples Northern U.S. Friday, August 15, 2003. http://www.foxnews.com/story/0,2933,94772,00.html (accessed 4 September 2005)

Ghiassi M., and Woldman K. I. S. (1994). Dual Programming Approach to Software Testing. *Software Quality Journal*, 3, 1994, Pages 45 - 58

Gilb T., Graham D. (1993) *Software Inspection*. Addison-Wesley. ISBN 0-201-63181-4.

Gilb T. (1997a) Requirements-Driven Management: A Planning Language. *Crosstalk*, Jun 1997, http://www.stsc.hill.af.mil/crosstalk/1997/06/requirements.asp (accessed 4 September 2005)

Gilb T. (1997b) Evolutionary Project Management. Proceeding International Software Quality Week Europe 97.

Gilb T. (1996) Requirements-Driven Management: A Planning Language. http://www.stsc.hill.af.mil/SWTesting/gilb.html
No longer available online. Link may become available from:
http://www.gilb.com/Pages/2ndLevel/gilbdownloadother.html#req-driv-mang

Glass R. L. (2004) The Mystery of Formal Methods Disuse. *Communications of the ACM*, Vol 47, No 8, August 2004, Pages 15-17.

Halstead M. H. (1977) *Elements of Software Science*. Elsevier.
ISBN 0-444-00205-7.

He Z., Staples G., Ross M., Court I., Hazzard K. (1997) Orthogonal software testing: Taguchi methods is software unit and subsystem testing. *Logistics Information Management*, Vol 10, No 5, Pages 189-194.

Hoffman D. M., Strooper P. (1991) Automated Module Testing in Prolog. *IEEE Transactions on Software Engineering*, Vol 17, No 9, Pages 934-943.

Hoffman D. (1999) Heuristic Test Oracles. *Software Testing & Quality Engineering*, March/April 1999, Pages 29-32.

IBM (2005a) Rational Rose Data Modeler.
http://www-306.ibm.com/software/awdtools/developer/datamodeler/ (accessed 4 September 2005)

IBM (2005b) Rational Test Realtime.
http://www-306.ibm.com/software/awdtools/test/realtime/features/index.html
(accessed 4 September 2005)

IEEE (1994) *IEEE Standard Classification for Software Anomalies*, IEEE Std 1044-1993. ISBN 1-55937-708-9, SH94399 (print) and ISBN 0-7381-0406-X, SS94399 (PDF).

Jeffries, R., Anderson, A., Hendrickson, C. (2001) *Extreme Programming Installed*, Addison-Wesley. ISBN 0-201-70842-6

JUnit (2005) JUnit.Org. Testing Resources for Extreme Programming
http://www.junit.org/index.htm (accessed 4 September 2005)

Juristo, N., Moreno, A. M., Vegas, S. (2004) Reviewing 25 years of Testing Technique Experiments. *Empirical Software Engineering*, Vol 9, No 1-2, Pages 7-44

Kit E. (1995) *Software Testing in the Real World.* Addison-Wesley.
ISBN: 0-201-87756-2.

Leveson N. G., Turner C. S. (1993) An investigation of the Therac-25 accidents. *IEEE Computer*, Vol 26, No 7, Pages 18-41.

Libes, D. (2005) The Expect Home Page. http://expect.nist.gov/index.html
(accessed 4 September 2005)

Lions J. L. (1996) Ariane 5, Flight 501 failure, Report by the inquiry board. Available from http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf (accessed 4 September 2005)

LogicaCMG (2005). Testing Times for Board Rooms.
http://www.logicacmg.com/pdf/tracked/testingTimesBoardRooms.pdf (accessed 13th October 2005).

Lutsky P. (2000) Information extraction from documents for automating software testing. *Artificial Intelligence in Engineering*, Vol 14, Pages 63-69.

McCabe T. J. and Watson A. H. (1994). Software Complexity. *Crosstalk*, December 1994, http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp (accessed 4 September 2005)

Mandl R. (1985) Orthogonal Latin Squares: An application of experiment design to compiler testing. *Communications of the ACM*, Vol 28, No 10, Pages 1054-1058.

Manolache L. I. and Kourie D. G. (2001) Software testing using model programs. *Software – Practice and Experience*, Vol 31, Pages 1211-1236.

Marick B. (1995) *The craft of software testing.* Prentice-Hall. ISBN 0-13-177411-5

Marick, B (2005a) Testing Foundations - Tools. http://www.testing.com/tools.html (accessed 4 September 2005)

Marick B. (2005b) The Testing Team's Motto. http://www.testing.com/writings/purpose-of-testing.htm (accessed 4 September 2005).

Mercury (2005a) Mercury WinRunner. http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/ (accessed 4 September 2005)

Mercury (2005b) Mercury QuickTest Professional. http://www.mercury.com/us/products/quality-center/functional-testing/quicktest-professional/ (accessed 4 September 2005)

Michael, C. C., McGraw, G., Schatz, M. A. (2001) Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, Vol 27, No 12, Pages 1085-1110.

Myres, G. J. (1979) *The Art of Software Testing*. John Wiley and Sons. ISBN 0-471-04328-1.

NERC (NORTH AMERICAN ELECTRIC RELIABILITY COUNCIL) Steering Group (2004). Technical Analysis of the August 14, 2003, Blackout: What Happened, Why, and What Did We Learn? July 13th 2004. ftp://www.nerc.com/pub/sys/all_updl/docs/blackout/NERC_Final_Blackout_Report_07_13_04.pdf (accessed 4 September 2005)

OSR (2000). You're Testing Me - Testing WDM/Win2K Drivers. *The NT Insider*, Vol 7, Issue 6, Open Systems Resources Inc.

Ostrand, T. J., Weyuker, E. J., Bell, R. M. (2005) Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, Vol 31, No 4, Pages 340-355.

Parzen, E. (1960) *Modern Probability Theory and Its Applications*. Wiley, New York. Reprinted as Wiley Classics Library 1992. ISBN 0-471-57278-0.

Phadke M. S. (1997) Planning efficient software tests. *Crosstalk*, October 1997

Poore, J. H., Walton, G. H., Whittaker, J. A. (2000) A constraint-based approach to the representation of software usage models. *Information and Software Technology*, Vol 42, Pages 825-833.

Poore, J. H., Trammell, C. J. (1998) Engineering Practices for Statistical Testing. *Crosstalk*, April 1998.

Poulsen K. (2004) Tracking the blackout bug. *SecurityFocus*, April 7th 2004. http://www.securityfocus.com/news/8412 (accessed 4 September 2005)

Prowell, S. J. (2000) TML: a description language for Markov chain usage models. *Information and Software Technology*, Vol 42, Pages 835-844.

Prowell, S. J., Poore, J. H. (2004) Computing system reliability using Markov chain usage models. *The Journal of Systems and Software*, Vol 73, No 2, Pages 219-225

Rankin, C. (2002) The Software Testing Automation Framework. *IBM Systems Journal*, Vol 41, No 1, Pages 126-139

Ross P. J. (1988) *Taguchi Techniques for Quality Engineering*. McGraw-Hill, New York. ISBN 0-07-053866-2

Savoye, R. (2005) DejaGnu. http://www.gnu.org/software/dejagnu/ (accessed 4 September 2005)

Segue (2005) SilkTest. http://www.segue.com/products/functional-regressional-testing/silktest.asp (accessed 4 September 2005)

SETI (2005) Seti@home. http://setiathome.ssl.berkeley.edu (accessed 23 October 2005)

Silberschatz, A., Galvin P. B. (1999) *Operating Systems Concepts*, Fifth Edition. John Wiley and Sons, Inc. ISBN 0-471-36414-2

So, S. S., Cha, S. D., Shimeall, T. J., Kwon, Y. R. (2002) An empirical evaluation of six methods to detect faults on software. *Software Testing, Verification and Reliability*, Vol 12, No 3, Pages 155-171

Software Research (2005) TestWorks. http://www.soft.com/Products/stwindex.html (accessed 4 September 2005)

Sommerville, I. (2004) *Software Engineering*, 7th Edition. Addison Wesley. ISBN 0-321-21026-3

STAF (2005) Software Testing Automation Framework (STAF) http://staf.sourceforge.net/index.php (accessed 4 September 2005)

Staknis M. E. (1990) Software quality assurance through prototyping and automated testing. *Information and Software Technology*, Vol32, No 1, Pages 26-33.

Symons C. R. (1991) *Software Sizing and Estimating*. John Wiley & Sons. ISBN 0-471-92985-9.

Tcl Developer Xchange (2005) Tcl/Tk. http://www.tcl.tk/software/tcltk/ (accessed 4 September 2005)

UML (2005) UML Success Stories, http://www.uml.org/uml_success_stories/index.htm (accessed 4 September 2005)

U.S.-Canada Power System Outage Task Force (2004). Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations. April 2004. https://reports.energy.gov/BlackoutFinal-Web.pdf (accessed 4 September 2005)

Vincenzi, A. M. R., Maldonado, J. C., Wong, W. E., Delamaro, M. E. (2005) Coverage testing of Java programs and components. *Science of Computer Programming*, Vol 56, No 1-2, Pages 211-230.

Vinter O. (1997) How to apply static and dynamic analysis in practice. *Software Quality Week*, Brussels, November 1997.

Viscarola, P., Mason, W. A. (1999) *Windows NT Device Driver Development*. Macmillan technical publishing USA. ISBN 1-57870-058-2.

Voas, J. M., Payne J. E., and Miller K. W. (1993) Automated Test Case Generation for Coverages Required by FAA Standard DO-178B. *Proceedings of Computers in Aerospace 9*, Oct 93, San Diego. Available from http://www.cigital.com/papers/download/aiaa_1_93.ps (accessed 4 September2005)

Voas J., Charron F., McGraw G., Miller K., Friedman M. (1997) Predicting how badly "good" software can behave. *IEEE Software*, Vol 14, No 4, Pages 73-83.

Vouk M. A. (1990) Back-to-back testing. *Information and Software Technology*, Vol 32, No 1, Pages 34-45.

Walton G. H., Poore, J. H., Trammell, C. J. (1995) Statistical Testing of Software Based on a Usage Model. *Software-Practice and Experience*, Vol 25, No 1, January 1995, Pages 97-108.

Walton G. H., Poore, J. H. (2000a) Generating transition probabilities to support model-based software testing. *Software-Practice and Experience*, Vol 30, Pages 1095-1106.

Walton G. H., Poore, J. H. (2000b) Measuring complexity and coverage of software specifications. *Information and Software Technology*, Vol 42, Pages 859-872.

Watson, A. H. and McCabe, T. J. (1996) Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Publication 500-235*, August 96, http://www.mccabe.com/iq_research_nist.htm (accessed 4 September 2005)

Welch, B. B. (1999) *Practical Programming in Tcl and Tk*, Third edition, Prentice Hall, ISBN 0-13-022028-0.

Weyuker, E. J. (1982) On Testing non-testable programs, *The Computer Journal*, Vol 25, No 4, Pages 465-470.

Whittaker, J. A. (2000) What is Software Testing? And Why is it So Hard? *IEEE Software*, Vol 17, No 1, Pages 70-75.

Whittaker, J. A., Poore J. H. (1993) Markov Analysis of Software Specifications. *ACM Transactions on Software Engineering and Methodology*, Vol 2, No 1, January 1993, Pages 93-106.

Whittaker, J. A., Rekab, K., Thomason, M. G. (2000) A Markov chain model for predicting the reliability of multi-build software. *Information and Software Technology*, Vol 42, Pages 889-894.

Whittaker, J. A., Thomason, M. G. (1994) A Markov Chain Model for Statistical Software Testing. *IEEE Transactions of Software Engineering*, Vol 20, No 10, October 1994, Pages 812-824.

Williams, T. W., Mercer, M. R., Mucha, J. P., Kapur, R. (2001) Code Coverage, What Does it Mean in Terms of Quality? *Proceedings Annual Reliability and Maintainability Symposium, 2001.* Pages 420-424.

Weyuker, E. J. (2004) How to judge testing progress. *Information and Software Technology*, Vol 46, No 5, Pages 323-328.

Zambonelli F., and Omicini A. (2004) Challenges and Research Directions in Agent-Oriented Software Engineering. *Autonomous Agents and Multi-Agent Systems*, Vol 9, No 3, Pages 253-283.